UNIVERSITY OF JOENSUU
COMPUTER SCIENCE
DISSERTATIONS 3

SIMO JUVASTE

# MODELING PARALLEL SHARED MEMORY COMPUTATIONS

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Science of the University of Joensuu, for public criticism in Auditorium M1 of the University, Yliopistokatu 7, Joensuu, on October 30th, 1998, at 12 noon.

UNIVERSITY OF JOENSUU
1998

| | |
|---|---|
| Julkaisija<br>Publisher | Joensuun Yliopisto<br>University of Joensuu |
| Toimittaja<br>Editor | Jorma Tarhio |
| Vaihto | Joensuun yliopiston kirjasto / Vaihdot<br>PL 107, 80101 Joensuu<br>puh. 013-251 2677, fax. 013-251 2691<br>email: Riitta.Porkka@joensuu.fi |
| Exchanges | Joensuu University Library / Exchanges<br>P.O.Box 107, FIN-80101 Joensuu, Finland<br>tel. +358-13-251 2677, fax. +358-13-251 2691<br>email: Riitta.Porkka@joensuu.fi |
| Myynti | Joensuun yliopiston kirjasto / Julkaisujen myynti<br>PL 107, 80101 Joensuu<br>Puh. 013-251 2652, 013-251 2662, fax. 013-251 2691<br>email: Armi.Lavikainen@joensuu.fi |
| Sales | Joensuu University Library / Sales of publications<br>P.O.Box 107, FIN-80101 Joensuu, Finland<br>tel. +358-13-251 2652 or +358-13-251 2677, fax. +358-13-251 2691<br>email: Armi.Lavikainen@joensuu.fi |

ISSN 1238-6944
ISBN 951-708-693-8
UDK 681.3.02
Computing Reviews (1991) Classification: C.1.2, D.1.3, F.1.2
Yliopistopaino
Joensuu 1998

# Modeling Parallel Shared Memory Computations

Simo Juvaste

Department of Computer Science
University of Joensuu
P.O.Box 111, FIN-80101 Joensuu, Finland
juvaste@cs.joensuu.fi

University of Joensuu, Computer Science, Dissertations 3
Joensuu, October 1998, 190 pages
ISSN 1238-6944, ISBN 951-708-693-8

Keywords: parallel computing, shared memory, modeling, F-PRAM

Interprocessor communication is the most difficult part of parallel computation on current parallel computers. Programmers find it difficult to correctly and reliably distribute and maintain the data of a parallel program. Most efficiency problems are due to excessive or inefficient communication. Parallel computer manufacturers find it difficult and expensive to build interprocessor communication networks that would keep up with fast processors.

In this thesis we shall present a new model of parallel computing, the F-PRAM model. The model characterizes parallel computers with a set of parameters, most of which model the limitations of the shared memory access of the processors, i.e., the communication. For the programmer, the new model offers a convenient abstraction of shared memory, but charges duely the machine-dependent costs of the use of the shared memory. For shared memory access, the model presents a new prefetching primitive. By using the model the programmer can avoid too expensive communication, and the parallel computer manufacturer can choose the most important features to improve.

The new model was tested with a fully configurable emulator of an abstract parallel computer. Using the emulator we implemented and analyzed a set of sample algorithms. These measurements revealed, e.g., the effects of insufficient shared memory access capabilities. Different algorithms tolerated different levels of scarcities such as insufficient bandwidth or high shared memory latency. We also estimated the values of the parameters on some existing parallel computers.

# Acknowledgments

This thesis is the result of the research carried out at the Department of Computer Science at University of Joensuu at 1992-1998. I wish to thank professor Martti Penttonen for introducing me to parallel computing and for supervising my research and this thesis. Also, the rest of our parallelism research group, Ville Leppänen, Martti Forsell, Anssi Kautonen, and Risto Honkanen have helped my research a lot via providing alternative but related views to parallelism. Pasi Hämäläinen and Jukka Veräjäntausta implemented parts of the emulator system. It was a pleasure to work with them, and I thank them for providing me a stable platform to work with.

Dr. Jussi Rahola and professor Per Stenström kindly accepted the role of a reviewer. I wish to thank for their efforts and for providing me valuable information on the details of my work. Also, I specially thank my colleague Ph.Lic. Stephen Eriksson-Bique for revising the language of the manuscript.

My work was financially supported by the Academy of Finland, Emil Aaltonen Foundation, and the Department of Computer Science and the Faculty of Science at the University of Joensuu.

The Department of Computer Science has been a pleasant environment to work at. The discussions with the staff of the Department have been great breaks in the middle of the work.

During the whole time of my study at the University, I have had dear friends with whom I have relaxed outside the study and work. I especially want to thank Olli and Pasi for all the fun time spent together. To this greatly enjoyable category of close friends, I also count the Finnish and Scandinavian nature. In addition to relaxation, also most of the best ideas for the work occurred to me while running or hiking somewhere out there.

My family provided the basis for my scientific career. I want to thank they especially for encouraging the attitude of pursuing for explanations for all answers.

I warmly thank my wife Hannele for sharing her life with me (and for tolerating me when I sometimes work late and go running even after that).

Life is fun!

# Contents

# Chapter 1

# Introduction

In this thesis we shall present a new model of parallel computing, an experimental testbed of the model, and a set of experimental data on the use of the model. Specifically, we model shared memory computation in computers with technology similar to current parallel computers. The processors compute independently using their local memories. The processors communicate via a limited asynchronous access to an amount of shared memory. The modeling concentrates on a set of parameters that describe the shared memory access limitations of the processors. The experimental part of the work was done using a configurable emulator to study various combinations of the different parameters.

In the following section we present the reasons to use a parallel approach for solving problems. Then we briefly discuss the current limitations and problems of parallel computing. Finally, we review the contents and the structure of this thesis.

## 1.1 Background

We program computers to accomplish different tasks. During the programming process, we usually describe the problem solving algorithm using some high-level programming language. Programming is usually easier if the structure of the algorithm corresponds in a natural way to the structure of the problem or the solution. In order to keep the programming efforts reasonable, the programming language being used should support structuring the program to follow the structure of the problem or the solution. For example, if some real world phenomena occurs concurrently, requiring the programmer to sequentialize the operations is a possible source of mistakes. The other important requirement for any programming system is that it must produce an efficiently executable program after compilation. Moreover, it would be advantageous if the program would be portable to different computers as easily as possible. These requirements can be somewhat conflicting, but in sequential programming the current programming languages provide a rather good and rather widely adopted compromise. These languages, nevertheless, produce sequential programs, which is an unnecessary restriction. In parallel computing, a good compromise for expressing parallelism remains to be found.

### Why parallelism?

To solve a problem we write an algorithm that evaluates the results using a set of simple atomic operations. If the number of required operations is large, we either need a lot of time to perform them, or a way to perform a lot of operations in a given time.[1] The time is usually limited by human factors, or by the forecasting nature of the problem. Thus, in some cases the operations need to be computed fast. The number of operations one processor can perform in a time unit is limited by the technology of the era. Still, waiting and buying faster and faster processors is currently the most popular way to speed up performance of a system. Because of the vast amount of money used for the development of personal computers and high-performance workstations, the power of the fastest microprocessors is doubling in less than every two years. Considering the requirement of a reasonable amount of time to complete the task, waiting for a faster processor to be developed is, however, out of the question as a general solution. A task that has to complete this week cannot be postponed for two years in order to be able to complete it then twice as fast as now. Furthermore, there will probably be, after all, some physical limits which will slow down the speed of the development of the processors. Overall, the possibilities of the uniprocessor approach to speed up a computation are rather limited. Consequently, we must use several processors together if we want more work to be done in a given amount of time.

### The use of parallelism

We use several processors to perform several operations simultaneously, i.e., in parallel. To use several processors for solving a single problem faster than with one processor, we can divide the problem to several subproblems, compute the subproblems in *parallel*, and finally combine the results.[2] If the division is reasonable and balanced, and if the division and combining will not use too much time, the resulting *parallel algorithm* will be faster than the original serial algorithm, which was our first goal.

The increased speed does not come for free. A machine with more than one processor will cost more than a machine with only one processor. The parallelized version of the algorithm should thus be significantly faster to justify the increased hardware costs. To be fully cost effective, a parallel computer with 1000 processors should be 1000 times faster than a sequential one. Using 1000 processors we can solve a problem at most one thousand times faster than with one processor. In other words, using one thousand processors, we can achieve at most 1000-fold *speedup*. If we could solve the problem more than one thousand times faster, i.e., *superlinearly faster*, the parallel algorithm would give us a new, faster serial algorithm. Any reports on superlinear speedups in some applications are due to some machine-dependent anomalies, such as increased cache size. Even if the speedup is constant, e.g., 1000, using any given parallel computer, the speedup still is greater than what it is possible to achieve by other means.

---

1. A third way would be to improve the used algorithm or its implementation. These, however, cannot be improved infinitely. Improvements in reasonable implementations of decent algorithms and matured compiler technology are seldom significant. Moreover, these tunings tend to be very labor intensive, and, thus, eventually rather expensive.

2. There are also other ways to compose parallel algorithms than *divide-and-conquer*.

Since the maximum available speedup is limited by the number of processors used in parallel, we must try to exploit the parallelism provided by the processors as efficiently as possible. In other words, as little as possible of the power of the parallelism should be wasted on other operations than the actual computing. The inefficiency induced by the parallelization should be a constant, and preferably a small fraction of the number of processors. An inefficiency that increases with the number of processors is generally not acceptable unless we seek only absolute speed regardless of the costs. For example, a logarithmic inefficiency ($\log_2 P$), i.e., a situation, where only $1/\log_2 P$ of the work is useful, would be 85% in a 100 processor computer, leaving us only the power of 15 processors. The most common cause of slowdown in problem solving is the time used in communication between processors. To be able to compute anything in parallel, the processors need to communicate in some way. The more we divide the task, i.e., provide more parallelism, the more the processors need to communicate. In most cases the communication is the most difficult part in the whole process of parallel programming. Also, the communication is the weakest point of most of the existing parallel computers. Consequently, the interprocessor communication is one of our main subjects in this thesis.

Besides communication, the other difficulty induced by the parallelization is the load balancing between the processors. Unless each processor has the same amount of work to do, the idling of the waiting processors reduces the efficiency of the whole execution. The balancing of the work is especially difficult in case of irregular data, e.g., in physical simulations. The load balancing problems are very problem dependent and they form a separate field of research. Consequently, we shall mostly ignore the load balancing issues within this thesis.

We concentrate on the speed and efficiency of computing since parallel computing is meaningful only for the most time-consuming and time-critical tasks. The smaller tasks can be accomplished using serial programming and serial computers. Many time-consuming real-world applications have rather high time complexities, often in the range from $O(n^2)$ to $O(n^4)$, where $n$ is the size of the problem.[3] The problems of linear time complexity are rarely too time consuming to require parallel processing.[4] An exception is the time-critical database operations, which include scanning, choosing, and gathering some information quickly out of a big database. On the other end of the time complexity spectrum, the NP-complete[5] problems are not very interesting either, since we can solve only marginally larger problems even if we use an order of magnitude more processors.

Some applications require that certain operations are executed seemingly simultaneously. For example, a user interface should be able to respond to user actions even while updating the display. Such a system is called *concurrent*, and it does not usually require genuine parallel execution of the different tasks, hence it can be accomplished by time-sharing a single processor. Concurrent programming has slightly different goals than parallel programming, and, thus, it is out of the scope of this thesis. We should note, however,

---

3. Informally, $O(f(n))$ stands for a growth rate of *at most order of* $f(n)$ [63].
4. However, the less time-consuming problems may occur as intermediate stages of larger problems. If a hard part of our program uses a lot of processors, then we should use the same number of processors for all parts of the same program if possible. Consequently, the parallel algorithms for linear time problems may still be useful.
5. Exponentially time consuming, according to the current knowledge.

that concurrent programs can be executed with several processors, and therefore with genuine parallelism.

## 1.2 Modeling parallel computation

The main topic of this thesis is the modeling of parallel computation. Most of the current research on parallel computing is based either on plain theoretical models of computing, or on some existing physical computer architectures. The theoretical models, e.g., the PRAM model, provide a bit too abstract view of the parallelism. Therefore, the resulting PRAM algorithms may not be efficiently executable on current parallel computers.[6] On the other hand, application programming for existing parallel computers usually results in rather unportable programs. Moreover, because of the lack of an applicable theoretical algorithmic background, the programs are hard to write, and do not always make the best use of the computers. Since parallel computing is vital especially for the most demanding applications, we must optimize our algorithms for both speed and efficiency of the computations. In other words, we should be able to design algorithms that yield the maximum power from a given parallel computer, and preferably also from the other parallel computers. Furthermore, we should be able to find the most important and cost-effective features of parallel computers to be able to develop cost-effective parallel computers in the future.

There have been several suggestions to fill the gap between the abstract algorithms and the existing parallel computers. Some examples of the proposed models are BSP [104], LogP [28], Y-PRAM [99], APRAM [23], and Phase PRAM [38]. Each of these models provides a set of restrictions and other features that should make the model more realistic. Most of these models still provide quite abstract views to parallelism, and the resulting algorithms are still quite fine-grained since not all of these models emphasize all the costs of the interprocessor communication. Moreover, we feel that the best possible set of model features has not yet been found. Especially, we consider a fully asynchronous shared memory model to be a good compromise between the ease of programming and the ease of implementation. Consequently, we shall present a new model of parallel computing, that models parallel computers more accurately, and includes a set of guidelines on the use of the model to make it easier to use. The contribution of the new model is that besides presenting a model of parallel computation, it also models parallel computers and parallel programming. Since the model is designed to model both different parallel computers and parallel algorithms, it supports the design of portable algorithms. In addition to the model, we shall give a preliminary algorithmic base consisting of algorithm design principles and sample algorithms.

Our main goals are efficient and realistic parallel computations. The realizability goal requires the model to be implementable. At the machine level this means that we should not make unreasonable assumptions on the technology and that we should take the machine building costs into account. At the programming level the realizability means reasonable programming efforts. Thus, one of the goals of this thesis is to find a good compromise between the computation efficiency and the programming efficiency.

6. The purpose of the PRAM in not to achieve efficient algorithms for current parallel computers. We shall discuss the nature of the PRAM model later in Chapter 3.

### The structure, scope, and novel proposals of this thesis

The rest of this thesis goes as follows. In Chapter 2 we shall discuss the technological background of parallel computing, and the guidelines the technology sets for the possible computation models. Chapter 3 presents some existing parallel computation models and their shortcomings. These two chapters form an extended introduction to the main subject, the new model of parallel computation.

As the main contribution of this thesis, we shall define the new parallel computation model in Chapter 4. Compared to the existing parameterized models our new model presents a new set of parameters and a new shared memory access primitive.[7] Moreover, the model allows more asynchrony and optionally dynamic change of the parameters. The asynchrony of the shared memory references probably simplifies the structure of the processing nodes. We shall not, e.g., separately model the use of caches. Also, the routing machinery needs not to take care of the sequential consistency of the shared memory accesses.

The model is an asynchronous parameterized shared memory model. Therefore, these attributes can be considered as the main scope of this thesis. As we use shared memory, the processors communicate through it. Since the communication is the most difficult aspect of parallelism, this thesis also focuses on the communication through shared memory, i.e., the access of the shared memory. Chapter 5 describes a programming model for the new parallel computation model. An important idea is also a fully configurable emulator system for the parameterized shared memory model. We shall present the emulator system in Chapter 6. Using the emulator, we have been able to measure the impacts of the different parameters accurately. We shall present the measurements and analyses of some example algorithms in Chapter 7. The outcome of these measurements include the critical requirements different algorithms set for the communication capabilities of parallel computers. To link the model to real life, we shall compare some of the existing parallel computers with the new model in Chapter 8. Finally, we shall make some conclusions and sketch future research in Chapter 9. The conclusions also include a relational schema of the main concepts of this thesis in Figure 9-1.

7. The primitive has previously been used for creating new concurrent processes, not for shared memory access.

# Chapter 2

# Overview of the technological background

The progress of technology sets limits for the building of parallel computers. Also, at least as important are the applications that are executed in parallel computers. Both factors must be taken into account since the value of a parallel computer is determined by the ratio of its power on a particular application to the cost of building and programming the computer. Especially, investments in technology are wasteful unless they lead to improved performance on applications. More generally, we should consider all technology costs against the benefit we gain from every feature. This price–gain relation will be one of the leading themes throughout this thesis.

To be able to model an existing parallel computer, we must have some knowledge of the structure of the computer. To be able to design a model for different types of parallel computers, we must have some knowledge of the basic building blocks of which computers are made. Since we cannot include all possible components and their features into any model, we must be able to choose the set of the most important features that affect the performance and the cost of computers.

In addition to modeling the existing parallel computers, we should be able to model future parallel computers to be able to analyse which features would be important. In other words, our research should not focus only on hardware created by parallel computer manufacturers. By studying the importance of the different features, we can make suggestions on future development directions for parallel computer manufacturers. Consequently, in addition to the knowledge of performance-related factors, we must know what can, and cannot be done with existing and forthcoming technology. Without this knowledge we might include some unrealistic features in the model. Using an unrealistic model in algorithm design can produce unrealistic algorithms. To put it literally, a computer that cannot be built is of theoretical interest only.[8] Within this thesis, however, we concentrate on computers that are not very different from current parallel computers. Especially, we shall not count on, e.g., optical communication as a solution for the bandwidth problem.

In this chapter we shall discuss rather general aspects of high-performance computing. We shall use the terms parallel computer and supercomputer since the technological problems in parallel multiprocessors and traditional vector supercomputers are mostly similar. Furthermore, the traditional supercomputers are highly parallel, even if the pro-

8. Naturally, this theoretical interest can still be of great value, especially if it can guide the future development of parallel computers.

gramming is serial, and the larger modern multiprocessor parallel computers definitely are supercomputers. When we specially mean the traditional vector supercomputers, such as the Cray Y-MP series, we shall state so.

In this chapter we shall discuss the technology of different parts of parallel computers. In Section 2.1 we shall present the basic hardware components and their relationships. In Section 2.2 we shall briefly present different approaches used in existing computers to achieve high performance via parallelism. We shall continue the analysis of the existing parallel computers in Chapter 8 using the new model of parallel computing. In Section 2.3 we shall present some existing special purpose and special structured parallel computers. In computing, software technology is at least as important as hardware technology, but we shall discuss it in more depth in Chapter 3.

## 2.1 The components of parallel computers

Each node of a parallel computer needs to contain, or have access to, the same parts as a serial computer to be able to compute anything by itself. The standard components of the serial computer are a processor, some amount of memory, and an I/O connection. In a parallel computer there are several nodes of this kind. To be able to work on a common problem, these nodes must additionally have a medium to communicate with each other. In the four first subsections of this section we will briefly discuss each of the above components. Balance of the components in any computer is important because in an unbalanced system the most underpowered component forms a bottleneck of the computations. We shall discuss the balance of the components in the supercomputers in Subsection 2.1.5.

### 2.1.1 Processors

In sequential computing, the only goal of a processor is to execute given instructions as fast as possible. Other factors, such as programmability, are less important, especially if we consider only selling the processor. In parallel computing the speed is still an important factor, but the processors must also be able to communicate with the other processors or the global memory without losing the speed. Later in this subsection we shall discuss some tricks that can be used to allow the processor to tolerate the latencies of communication.

The means of designing faster and faster microprocessors have been the same ones as we presented in the introduction, namely faster clock speed and more parallelism. Concerning the clock speed, the current (early 1998) headlines of the computer magazines refer to clock speeds at 300-1000 MHz. The other, less flashy but at least as important, trend in the development of microprocessors is to exploit more and more parallelism in an apparently serial processor. This is a natural development since the speed of the logic gates cannot be increased very much anymore. Furthermore, as the cost of one gate decreases as the integration degree increases, there are more gates available in a chip to exploit the parallelism.

Even if both the increased clock speed and the increased parallelism seem to provide more and more processing power, they also provide more problems. At clock cycles close to 1 ns, the locality of all operations is vital. Even if light travels 30 cm in 1 ns, the signals on the semiconductor, aluminum, or copper interconnections cannot travel this fast. For example, at 600 MHz the ALUs of the Digital 21262 cannot access a cache on the other edge of the same silicon chip in one clock cycle. As we exploit more parallelism within the chip, the parallel components of the processor also have to communicate more. And the more there are parallel components, the slower the communication is. The decision between a short clock cycle and large-scale parallelism is thus a compromise for the execution of a sequential instruction stream. The fastest clock cycles can be achieved with the simplest processors, not necessarily achieving the fastest overall performance [93].

Here we shall present the two most common possibilities to introduce parallelism within microprocessors. The modern pipelined reduced instruction set (RISC) microprocessors are able to execute nearly all instructions apparently in just one clock cycle. The word "apparently" stands for the fact that even if the instructions take longer than one cycle, a new instruction is initiated in every cycle. As different execution stages of several consecutive instructions are executed concurrently, the processors are said to be *pipelined*. The fashionable marketing-term "superpipelining" stands for using a deep, typically 6-9 stages, pipeline. The advantage of deep pipelines is that the clock cycles can be shorter since the execution of the shorter stages of instructions is faster. Generally, the maximum clock-rate of a processor is determined as the product of the gate-delay of the used technology and the longest path of gates in a single pipeline stage. The disadvantage of pipelining is that the successive instructions often have dependencies, which prevent the execution of the latter instruction until the result of the former one is available. Furthermore, the successor of a branch instruction is not known in advance, but has to be guessed somehow. If the guess was wrong, the pipeline must be emptied from the wrongly started instructions and the correct instructions have to be started from the beginning of the pipeline.

Since improving the processor instruction rates by increasing the clock-rates will be more and more difficult, the processor designers have begun to exploit *superscalar* execution of the instructions. In a superscalar processor more than one instruction can be issued at the same clock cycle.[9] Modern microprocessors have 5-10 independent execution units and can issue up to five instructions simultaneously. The disadvantage of the superscalarity relates to the same problem as with pipelining, i.e., the instruction dependencies. If the successive instructions are dependent, they cannot be issued concurrently. The best method to circumvent this problem is to use an optimizing compiler that tries to generate code which has less dependencies. Processor techniques for solving the problem include reordering the instructions and using direct connections between the instructions instead of using the registers.

As we noted above, the added parallelism, both superscalar and pipelining, complicates the coordination of the execution of the instructions. An especially difficult goal is to keep all the resources of the processors active. The optimizing compilers will have a big responsibility in this regard in the future, otherwise processors will work with only a fraction of the maximum possible power. The extreme development direction in parallel processors are the very long instruction word (VLIW) processors. A VLIW-processor executes instructions that consist of several, say 16, independent simple instructions.

9. The instruction stream generated by the compiler is, however, sequential. The processor analyses the instructions and executes them in parallel if possible.

Unlike with the superscalar discussed above, the composition of the long instruction, i.e., the parallelization of independent instructions, is left as the responsibility of the compilers. Another feature, which is left to the compiler is the minimization of the use of slow memory. Since the processor clock-rates have speeded up faster than memories, the modern architectures support only register-to-register operations. All memory accesses must be done using separate load and store instructions. Even if the first-level caches were fast enough to provide the data in one clock cycle, the registers work better in delivering data between consecutive instructions in pipelined processors. For current microprocessor systems, the optimization of the use of memory and caches is perhaps the most vital single issue when pursuing for optimal or near optimal performance.

Perhaps the most notable trend in current microprocessor development has been the extremely high development costs of new processors. In the near future, few companies can afford the development of a new competitive processor even using an old instruction set architecture (ISA) [85]. To distribute the high initial costs, the manufactures need to be able to sell millions of the new processors. Consequently, the future parallel computers will probably continue the current trend of using off-the-shelf workstation microprocessors. The problem with this trend is that the processors are poor communicators since they were designed to be used in workstations with only single or few processors. As it is today, there will probably be some slightly modified versions of microprocessors available in the future, but as the initial costs of manufacturing grow, the special versions will be more expensive. Moreover, the modified versions are usually of an older generation.

Most multitasking operating systems trigger a context switch in case of a I/O request since the processor can execute at least tens of thousands of instructions during the request, while a context switch takes only, e.g., a hundred clock cycles. In parallel computers, a similar occasion arises during shared memory requests, which take a considerably long time compared to the instruction execution time, typically hundreds of clock cycles. The time required for a context switch just has to be cut down to a few clock cycles. The current standard microprocessors cannot do this, but some research processors can. In practice, the processor must be able to store multiple register sets within it. An example of this approach is the Sparcle processor, which uses the register windows of a Sparc processor as storage of the registers of the different contexts [1]. Tera computer company is producing a commercial parallel computer based on this approach [8]. They take the multithreading approach to the extreme by allowing the processors to change process, or the execution thread, after every instruction. A big advantage of this approach is that consecutive instructions are totally independent of each other, hence the processors can have very deep pipelines [33]. Furthermore, the processors would not need caches to local memory, since the local memory latency also gets hidden by the multithreading system. The problem of this approach is, however, the amount of threads required to completely hide the time needed to reference the shared memory. Furthermore, the interconnection network must be able to handle the memory references made by potentially all threads.

Traditional supercomputers have used vector processors to improve the performance on scientific computations. A vector processor is able to execute the same operations to the elements of one or more vectors at a rate of one element per clock cycle. The execution is based on very deep pipelining and a set of vector registers, which typically have space for 64 elements. The memory requests are pipelined over the vector so that the vector registers are filled and saved on background while the actual execution works on other elements. A typical vector processor includes 2-6 vector pipelines, each of which is able to issue a floating point operation, or a multiply-add operation, on each clock cycle. Because of the deep pipelining and because the consecutive operations are independent, the clock cycle can be very fast. To keep the pipelines filled, the processors need a very high-capacity memory connection. A two-pipeline processor requires four words to be loaded and two saved for every clock cycle. The speed of memory connection has been the strength by which traditional supercomputers have often beaten the modern "killer workstations." Many of the earlier parallel computers have also had a couple of smaller vector processors embedded to each processing node since the earlier off-the-shelf microprocessors have had rather poor floating-point performance. The boosted processor nodes have had quite good theoretical peak performances, but rarely have been very usable for real applications because of the limited memory and communication bandwidths. Modern RISC-microprocessors have good floating-point performance themselves, consequently the separate floating-point processors are disappearing.

### 2.1.2 Memory

The available memory capacity per chip and per dollar[10] is increasing nearly as fast as processor speed. On the other hand, memory speed is increasing only slowly since the manufacturers currently optimize the new DRAM memory chips for capacity, not for speed. On the other hand, faster memory technologies, such as SRAM, will remain a couple of times more expensive than DRAM. Since the memory makes a considerable part (tens of percents) of the price of a computer, the faster memory technologies can be applied only to the most expensive computers, or to special purpose computers, which might require only a small amount of the fast memory.

Because the performance gap between the microprocessors and reasonably priced (DRAM) memory is still increasing, more and more sophisticated hierarchical local memory, i.e., caches, are needed. New microprocessors are designed to operate with 2- or 3-level caches to minimize the frequency of the accesses to the slow main memory and to minimize the time required to fetch data from the most innermost cache. The bigger the cache is, the bigger part of the memory requests it can serve. The natural problems of the big caches are the cost and bigger physical size. A smaller cache can be built with faster access times, with more access ports, and with more logic than a bigger one. Additionally a smaller cache can be located physically very near to the processor, i.e., on the same chip. The 2- or 3-level cache structures form combinations between these features.

The problem of all caches is that there are applications that frequently use more memory locations than fast caches can provide. Or worse yet, there may be random references to a large set of data locations, causing all memory references having to be served from the main memory with a considerable latency. Aids for this latency are block fetching and prefetching. In other words, we can fetch a bigger block of main memory to the cache than required by the single request. The fetching of the bigger block can be pipelined to exploit burst transfer protocols. The advantage is that we can fetch the data of sev-

10. As opposed to other development in silicon industry, the memory prices, however, decrease and rise quite irregularly.

eral requests with only one latency. Naturally this will not work for totally random accesses to the main memory. Prefetching can be used if the processor is able to examine instructions in advance and to issue the memory requests in advance. The problem is, however, that the prefetching should be done tens of clock cycles before, and no processor can predict reliably that far. The current microprocessors do not exploit the prefetching well in comparison with the traditional vector supercomputers. The prefetching can, however, be implemented also by a separate entity independent of the processor. The processing nodes of the Cray T3E have dedicated hardware for monitoring the memory references[11] made by the processor. If the processor accesses several memory locations sequentially, the support circuitry starts a pipelined prefetching of the subsequent words [26]. The general solution for accurate prefetching would be guiding via compiler and/or programmer directives.

No matter how smart prefetching technique is used, the bandwidth in every stage between the processor and the main memory must be big enough to provide data to the processor fast enough. For example, the dot product of two vectors of 64-bit floating point numbers requires 16 bytes to be transferred for each multiplication-addition pair. Since the two operations can be executed in parallel (or pipelined with the next pair) in a modern superscalar processor, we would need 4.8 GB/s memory bandwidth for a 300 MHz processor. Because few current systems can do this, the real performance for these real applications is considerably poorer than the theoretical peak performance of the processors.

Like speed, the size of the memory affects real applications more than the benchmarks. As we stated earlier, parallel computing is useful mostly for jobs that are so demanding that sequential computing would be prohibitively slow. Such jobs tend to demand not only a lot of processor time, but also a lot of memory. For example, in many simulations the functions of time and space requirements grow equally fast as the accuracy improves. Consequently, relative to the processing power, parallel computers used for such jobs should have at least as large memories as sequential computers have. In many cases, the algorithms parallelize better the bigger problems are. That is, we can get more GFLOPS out of the computer as we increase the memory. One example is the LINPACK benchmark where the vector lengths can be scaled up to get the maximum power [29]. Having enough memory is especially important in parallel computers to keep the processes from encountering unexpectedly long delays because of page-faults, and, thus, causing severe interprocess synchrony problems. Actually, any paging to and from a magnetic mass storage is too slow for high-performance computation.

With a lot of memory in each processing node, we can hide some communication and input/output -bottlenecks by having several concurrently executing jobs and by switching to another task in a case of a pagefault or other delay. This naturally causes even more severe interprocess synchrony problems, but may be acceptable in some tasks having less tight synchrony.

### 2.1.3 Communication media

One of the fundamental reasons why parallel computing has been so underestimated is that communication capabilities of the parallel computers have not kept in pace with the

11. More accurately, memory references that cannot be fulfilled from the caches.

processors. As the communication capacities are poorer than the computation capacities, naively parallelized programs rarely achieve reasonable performance. Meanwhile, the communication media of the traditional vector supercomputers have been developed along with the processors. Consequently, many users have held to vector computers or switched to cheaper workstations.

The task of the communication medium is to deliver messages between the processors with low latency and large bandwidth.[12] In theory, this could be best accomplished using all-to-all connections, i.e., a complete graph, between the processors, but the number of connections in a complete-graph network is nearly the square of the number of processors. Furthermore, the degree of each processor would be linear to the number of processors. Electronic devices cannot have unlimited fan-ins or fan-outs. Thus, each bigger-degree node would have to have intermediate stages between the network connections and the processing node. The crossbar-switch topology solves the fan-out and fan-in problems, but it requires a quadratic number of routing switches, and makes queueing of concurrent messages more difficult.

Since all-to-all networks are too expensive for large scale parallelism, we have to settle for networks with lower degree and nonconstant diameter.[13] The physical wires and switches are configured by some fixed *topology*, i.e., they form a fixed graph where the wires are the edges and the switches are the vertices. There are plenty of possible topologies available to be chosen from when building a parallel computer. The interconnection networks are usually divided in two classes, *single stage* and *multistage* networks [92]. A single stage network connects the adjacent processors directly together with no adjacent stages. On the other hand, a multistage network has intermediate, routing-only nodes between the processing nodes. The advantage of the single stage networks is that they do not require any additional nodes besides the processing nodes. The processing nodes naturally need to have the routing logic which routes the messages. The nodes and the interconnections need to route also messages not belonging to them. A typical multistage network, such as the butterfly, has a logarithmic number of stages, and hence $P \log P$ intermediate nodes for $P$ processing nodes.

The most usual single stage networks are meshes, as used, e.g., in the DOE ASCI Red [90], tori, e.g., Cray T3E [25], and hypercubes, e.g., NCUBE [44]. The SGI Origin [67] uses a complex combination of SMP nodes, crossbars, and hypercubes. The most usual multistage networks are butterflies, omega networks, and fat trees, as used e.g., in the Thinking Machines CM-5 [98]. Some instances of these networks are shown in Figure 2-1. Many popular graph topologies have a very small logarithmic diameter. In a real parallel computer, the network must be embedded to the three dimensions. Any graph can be embedded in three dimensions [43], but we consider the physical sizes of the vertices and edges. Consequently, the volume of any machine is linear and the diameter is $\Omega(\sqrt[3]{P})$ in a three dimensional space.[14] Furthermore, some of the edges also must have $\Omega(\sqrt[3]{P})$ length [107]. Another physical problem of some attractive networks, especially the hypercube, is that the degree of the network increases, usually logarithmically, as the

12. In a shared memory system the messages are delivered between the processors and the shared memory.

13. The maximum number of routing steps to deliver a message between any two nodes.

14. Informally, $\Omega(f(n))$ stands for a growth rate of *at least* order of $f(n)$ [63].

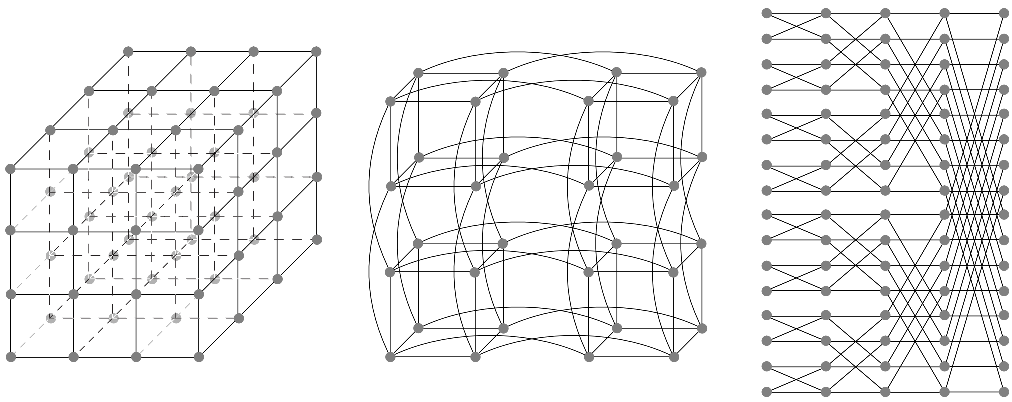

**Figure 2-1:** A 64-node 3D mesh, a 32-node binary hypercube, and an 80-node butterfly (with 16 input/output nodes).

number of nodes increases. Because the fan-out of any circuitry is limited, the high-degree nodes need to have some intermediate stages that may slow down the connection.

A compromise between the single- and multistage networks are the *sparse networks*. A sparse network consists of a large single stage network, most of whose nodes do not have a processor but only a routing switch. For example, a *P*-processor Tera computer system has a $P^{3/2}$ -node interconnection network where the processors, memory modules, and the I/O-nodes are located evenly on some of the nodes [8]. A *coated network* consists of a single stage network, such as a 3D mesh, but the processing nodes are located only at the outer nodes of the network, and the inner nodes are dedicated to the routing of the messages. Another type of the "hybrid" network topologies are the *meshes of buses* (MOB). A $d$ dimensional MOB with $P = p^d$ nodes resembles a $d$ dimensional mesh, but the node-to-node connections are replaced with buses of length $p$ in each dimension. The MOB has thus $dp^{d-1}$ buses. The diameter of the MOB is extremely low, only $d$, which usually is 2 or 3. The problem is that the buses will get crowded as $p$ increases. The capacity of the bus can be increased using several, or even $p$, channels on the bus using, e.g., optical connections and wavelength multiplexing, but that naturally also increases the cost. An interesting feature of MOBs is that most of the conventional networks can be efficiently embedded in MOBs, which, however, naturally have to have enough channels to support the degree of the network [70, 88].

The above considerations are valid for traditional electronic connections. Optical connections between the components can make a substantial difference compared to the electronic ones. The main advantage of using light instead of electric current is that the overlapping beams of light do not interfere each other. Consequently we can place several signals in the same fiber using different wavelengths. Moreover, light can be used to communicate through free space without any intermediate medium. The crossing beams will not disturb each other even if they used the same wave-length. Furthermore, the advances in the development of tunable transmitters can make it possible to have a moderately sized all-to-all connections in a single bus. In addition to the concurrency advantage, it is possible to modulate more data to a beam of light than to an electronic connection. The difference is big especially on long distances. The disadvantages of using optical connections are the problems in the connections between the electronic and optical components and the rather high price of the components. The optical technology is, nevertheless, improving rather rapidly. Thus, the optical communication may be a very good choice for the interprocessor communication in the future. Within this thesis, however, we concentrate on the approximately current level of technology. Consequently, we usually assume traditional electronic connections.

A real network consists of not only the physical wires and switches, but also a routing algorithm and a protocol to route all packets to their destinations. The routing algorithm defines the route, or the method to choose the route, to be used for delivering a packet to the destination. Usually the routing algorithm is realized in the communication nodes of the interconnection network. An *oblivious* algorithm uses only the destination information of each packet to choose the route. An *off-line* algorithm can use the information of all packets to be delivered to choose the routes. Off-line algorithms generally give better results, but they cannot be used in a general purpose parallel computer since it would require central or global control or knowledge of the routing task. The protocol also defines the routines for possible conflicts, for example, what to do when two packets should be sent to the same channel simultaneously.

### Synchronization network

The processors of modern parallel computers operate asynchronously. To be able to cooperate they need to be able to synchronize during the computation. The cooperation is realized by communication, which requires some synchrony between the processors. Message passing programs synchronize through the messages, but shared memory programs must have a separate synchronization facility. Synchronization can be accomplished either using the communication network, or a dedicated fast synchronization network. Some existing parallel computers have a dedicated synchronization network tightly connected to the processing nodes.

The explicit synchronization can be defined to be performed among two, several, or all of the processors. Pairwise synchronization is used in message passing computation, and it is easily defined by the fact that both processors know the other. In a shared memory model we, however, concentrate on the data, not on the processors or the messages. Synchronizing a dynamically determined set of processors is the most powerful synchronization primitive, especially if we are allowed to perform several disjoint synchronizations concurrently. Defining the synchronizations of arbitrary sets of processors is, however, rather difficult. In a reasonably small computer we could have a synchronization processor that processes the sets and broadcasts the synchronization signals to the slave processors. This approach is not very scalable, particularly when the number of sets increases. Alternatively, the processors could issue tagged synchronization operations, and then get synchronized according to the sets defined by the tags. This would also require central processing of the sets. Without the central processing capability, the processors should gather the shared or distributed knowledge of the synchronization sets, process the knowledge, and distribute the knowledge. The synchronization would require either efficient shared memory or a lot of pairwise messages. Furthermore, it might require the processors to be polled on whether they are going to participate in the synchronization or not. If the sets of the mutually synchronizing processors would be more regular, for example, hierarchical in powers of two, the partial synchronizations would be easier, especially if the machine had a divisible synchronization network of similar structure. This was suggested in the

*YPRAM* model [46]. The power of two sets do not, however, provide general sets for general computations.

In most data-parallel programs all processors perform the same computation on different data. During, or possibly between, the local computations, the processors use the shared memory to communicate and possibly to do I/O. The computation proceeds in phases that correspond to the algorithm and the decomposition of the data. Unless we use the SIMD-paradigm, the processors work asynchronously within each phase. Between the phases the processors have to synchronize to ensure the correct order of the executions of the different phases on different processors. Since all processors are working on the same problem, and presumably consume approximately the same time,[15] we can synchronize all processors at once instead of trying to determine the minimum required set of processors to be synchronized.

### 2.1.4 Input and output connections

The input and output of the data for parallel computers is rather much ignored by most algorithm researchers on parallel computing and by parallel computer salesmen. The salesmen of the parallel computers ignore it because they concentrate on the more impressive figures, notably on the peak processing power of the computers. The researchers probably ignore it because different parallel computers have rather different approaches for input/output, and because the theoretical models do not model I/O at all. Besides loading the input and saving the results, the I/O possibly includes more local operations such as paging of virtual memory and swapping stopped tasks to and from disks.

#### Input/output data

When considering the loading of data and saving the results, there are usually several stages of data transfers. Usually the input data comes from a single source. Thus, the whole system has a serial bottleneck even if the computer has parallel I/O capabilities. In a typical arrangement the user of the supercomputer transfers his or her data to a mass storage of the computing centre using either tapes or a slower network connection. Before the task is going to be executed the data gets transferred to the disks of the parallel computer, and from there to the memory of the computer at the beginning of the computation. The saving of the results gets done the opposite way. Supercomputers usually have an autonomous I/O system to transfer the data from the outside network connection or tape device to the local disks. Consequently, that phase does not disturb the processors of the computer. Thus, its speed is not important as long as the throughput is big enough to load enough tasks to keep the computer utilized. Because the transfer from the local disks to the memory of the computer can also be autonomous, the computer can pipeline saving a completed task, computing another task and loading the next task as long as the memory is big enough. We can conclude that among the autonomous I/O stages there exists a point[16] of the smallest width of the I/O system, which determines the maximum data rate to the computer. This data rate determines how much data we can transfer in a given time,

15. Assuming the parallelization is reasonable.

16. Or, more accurately, a "horizontal bisection."

which determines what kind mix of I/O intensive and less I/O intensive jobs we can execute without having the computer wait for the I/O.

In dedicated-use computers the I/O could be arranged with direct parallel links from the data source to the processors and from there to the data consumer. This approach is probably most useful in digital signal processing (DSP) systems, where a lot of data is analysed in real time. Typically the signal from a digital (video) camera is processed and analysed by an array of processors. In such a case each of the processors could have a connection to the corresponding segment of the CCD element of the camera.

**Local mass storage input/output**

Besides the actual I/O before and after the tasks, also some input and output during the tasks is required if the physical memory of the computer is not big enough to hold all the intermediate results of all of the ongoing tasks. This can be accomplished either using a virtual memory system or explicitly saving the intermediate results to the mass storage. Traditional vector supercomputers do not usually have virtual memory since it would cause extra complexity to the memory system, and, thus, slow it down. Moreover, the stalls induced by random page faults would corrupt the synchrony and load balance between the processors and probably ruin the efficiency.

The access times of the rotating magnetic disks are about a million instructions, which is an unacceptable time to wait. Therefore, a disk access causes usually a context switch, which also takes some time, especially in the vector processors. Traditional expensive supercomputers exploit solid state disks (SSD) in addition to the magnetic ones. The access time of SSDs is much more acceptable compared to the magnetic disks, which helps to avoid context switches. Even if SSD is much more expensive than magnetic disks, it is cheaper than high-performance memory because of the slower speed and lower bandwidth access ports.

Modern hard disks are rather cheap, small, and low power-consuming devices. Hence, there can be a local disk in each processing node of a parallel computer for the local operations such as swapping and paging. An existing example of this practice is the IBM SP/2 parallel computer, whose processor cards have one or two embedded hard disks [2]. The advantage of the local disks is that the I/O bandwidth scales up automatically as the number of processors increases without using expensive high-performance central mass storages. Furthermore, the I/O to the local disks does not consume global communication bandwidth.

Even if the processing nodes had their own local disks, the computer needs also a central mass storage. Most parallel computers rely solely on one (or few) high-performance disk arrays. To improve both bandwidth and capacity simultaneously, the disk arrays distribute (stripe) the data to several disks. The drawback of this arrangement is that concurrent distinct mass storage requests will be sequentialized anyway. For example, the references to the local temporary data of each processor will be slower than in a distributed disk system.

### 2.1.5 Balance of the components of supercomputers

Case and Amdahl have proposed rules of thumb on the requirements on the relative requirements on the different subsystems of computers [45]. The combined rule states that a balanced computer system has one megabyte of memory and one megabit/s of I/O bandwidth for each MIPS of processing power. The rules were valid at the time they were proposed and on machines having power on approximately that class. The rules appear to be scalable. In fact they propose linear requirements on the memory and the I/O capacity. Most of the supercomputer-class problems, however, have superlinear time complexities and only linear memory complexities. Consequently, the problems have sublinear memory and I/O requirements compared to the processing power requirement. Hence, instead of using the above traditional rule, we should examine more accurately the memory and I/O requirements of the problems when building or choosing a parallel computer. Later in this thesis we shall make such estimations using the new model of parallel computing. The development of changing the balance has been going on already for some time since the memory and I/O bandwidth prices have not decreased as fast as the processor prices. Consequently, the use of the above rule would cause modern supercomputers to have extremely expensive memories. For special purpose computers the ratios can be calculated based on the problem in question. For example, the DOE ASCI Red computer [74] has 585 GB of memory, 32 GB/s of I/O speed and 1.8 TFLOPS peak processing power, resulting in a MFLOPS:MB:MB/s ratio of 56:18:1, respectively. The I/O speed of the ASCI Red is probably not very important as the computer is mainly targeted towards very demanding and long-lasting tasks.

## 2.2 Classifications of existing general purpose parallel and high-performance computers

Parallel computers are designed and built according to some specifications and some purpose in mind, not according to any classification. For referencing purposes, however, we have to classify them. In addition to the architectural differences, the following classification can be considered as a classification according to the degree of the parallelism.

#### Networks of workstations (NOW)

The easiest way to create a parallel computer is to connect a group of stand-alone sequential computers with some communication medium. There exists a continuous spectrum of different degrees of tightnesses of this connection. At one end, the computers are connected to a shared memory using a crossbar switch, as it was done in the CMU C.mmp parallel research computer [109]. At the other end are the Ethernet or Internet connected distributed computing systems, called *networks of workstations* (NOW). The communication between the computers is often programmed using some of the portable programming libraries, such as MPI, which we shall briefly present in Section 3.4. These libraries present a common factor between the NOW and the distributed memory massively parallel computers.

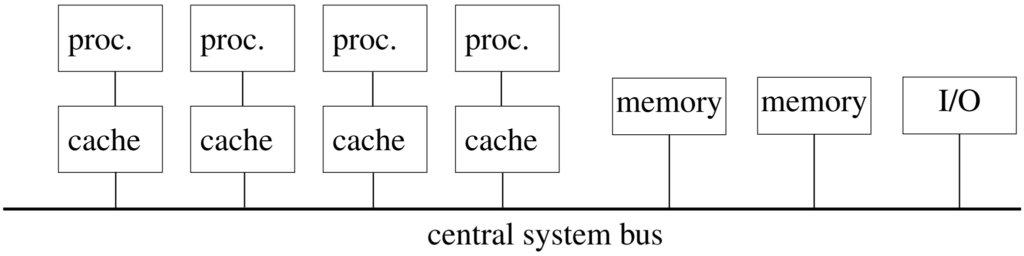


**Figure 2-2:** Bus-based SMP computer.

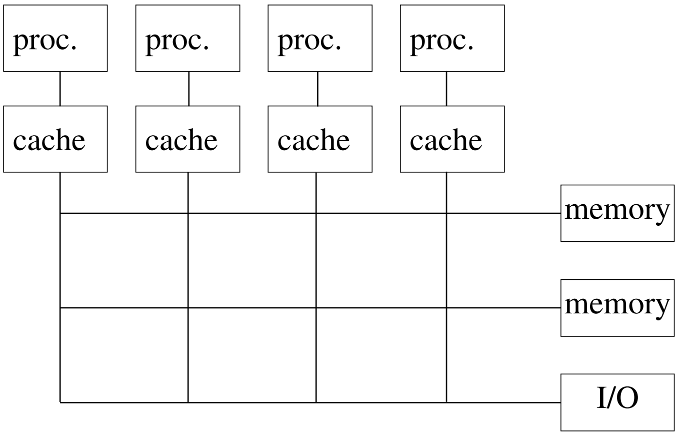


**Figure 2-3:** Crossbar-based SMP computer.

The wide variety of the distributed computing systems makes their classification difficult, but in general their characteristics and usability depend mostly on the ratio of the communication and computation capabilities. Moreover, the applications that are implementable on the NOWs are differ in their nature.

### Shared memory multiprocessors (SMP)

The easiest way to introduce parallelism into a single computer has been to connect several identical processors via a common bus to the same (banked) shared memory. This approach is called *symmetric multiprocessing* (SMP), see example in Figure 2-2. The bottlenecks of a SMP computer are naturally the single bus and the shared memory. As the processors speed up and require more memory bandwidth, the bus cannot serve very many processors fast enough. The traditional method to overcome the problem has been large local caches on each of the parallel processors to reduce bus traffic. The cache systems monitor the bus to be able to detect the possible updates made by other processors to memory locations they contain. This maintenance of cache coherence also consumes some of the available bus bandwidth. In spite of the big caches, the SMP computers will run out of memory bandwidth if the applications exceed the cache sizes. Nowadays most of the workstation and server manufactures provide multiprocessor versions of their top-of-line products. Typical configurations contain 2-8 processors and 1-4 memory modules on the same bus. As an improvement over the bus, newer SMP machines, e.g., the Sun Ultraservers, have more advanced topologies, e.g., a crossbar switch as in Figure 2-3, instead of the single bus between the processors and the other devices. This type of configuration makes the SMP architecture approach the classic supercomputers.

The SMP approach is usually used to increase the throughput of the computer instead of introducing real parallelism for the applications. Typically an SMP system is used to serve several interactive users and possibly a more computation intensive task concurrently without a severe slowdown. Because of the sequential nature of the small-scale SMP computers we shall not consider those much in this thesis. Moreover, a bit different types of models suite better for these computers.

Parallel applications for the SMP machines have been written either providing compiler directives and using a parallelizing compiler, or using a proprietary SMP library for each machine and operating system or using the portable message passing libraries. This lack of portable tools for parallel shared memory programming has probably been the reason for the slow development of SMP parallel applications. Recently, an industry group suggested a shared memory standard named OpenMP [83] as a cure for this problem.

Throughout the history of the SMP computers there have been clusters of SMP computers to elude the limitations of the SMP approach. The members of the SMP clusters share, e.g., the mass storages and the operating environment. The nodes are connected via a high-performance local area network (LAN) or via a specialized switch. Using a proper (message passing) programming environment the cluster can be used to solve larger tasks, but most often the clusters are used for solving separate tasks simultaneously. Because of the diversity of the SMP clusters, we shall not discuss them much within this thesis.

**Vector supercomputers**

Traditional supercomputers have used vector processors, which we presented briefly in Subsection 2.1.1. Additional parallelism on supercomputers is similar to the SMP approach, i.e., a couple of processors are connected to the same memory. The difference has been that on supercomputers the memory connection has not been implemented using slow buses, but extremely fast multistage switches which are able to serve all the processors even if they all make a lot of memory references. Consequently, the processors have be used also in parallel for one task. Still, a large portion of the use of the several processors has been to improve throughput. Hence, we shall not consider the vector supercomputers much in this thesis either. The Fujitsu VPP700 supercomputer, like its predecessors, is a hybrid of the traditional supercomputers and multiprocessor parallel computers [102]. It can have up to 256 processing units connected with a crossbar interconnection network. The largest VPP700s built so far has, however, contained less than a hundred processors.[17] The vector supercomputing nature of the VPP700 lies within the processing nodes. Each of the processing units can have up to 16 concurrent vector pipelines and 2.2 GFLOPS peak processing power.

17. Not counting the unique 166-processor NAL NWT 2 computer, which uses mostly the same components but is tailored to a dedicated use [79].

### Massively parallel computers (MPP)

To distinguish from the previous classes of computers, the large-scale parallel computers are usually called *massively parallel processing* (MPP) computers. The MPP computers can be divided into two classes. The first class includes computers with tens of thousands of very simple processors. The second class includes computers with tens to a thousand of high-performance microprocessors. The other traditional classification is the division to *single instruction, multiple data stream* (SIMD) and *multiple instruction, multiple data stream* (MIMD) computers proposed by Flynn [30]. The two classifications produce similar divisions of the existing parallel computers.

The commercial SIMD computers, such as the Connection Machine [50], typically have several very simple processors integrated in a single chip and connected with hypercube or mesh structured connections. The single processor usually computes with only one or a few bits, but using several processors together, also longer words can be used. Connecting together several of these single chip[18] modules of a few processors, computers having tens of thousands processors can be made with reasonable expenses. The processors are coordinated using a central control processor, which resolves the stream of instructions and broadcasts the same instruction for each of the processors. The processors have a set of special registers by which they decide whether to execute the given instruction or not. The advantage of the SIMD approach is that the processors can be extremely simple since they need no logic for branching and other program flow operations, and no memory for the program. Furthermore the processors are automatically synchronized by the single instruction stream. The disadvantage of the SIMD approach is that since the processors cannot branch, part of the processors have to wait idle during the conditional branches of the program. Furthermore, the floating point performance of the single-bit processors have been rather poor. Also the global broadcasting of the instructions limits the clock speeds, which is probably the reason why the massively parallel SIMD computers are disappearing. The best applications for the SIMD have been massively parallel databases, text searches, and expert systems.

The literal difference between the SIMD and MIMD approaches is that each of the processors of a MIMD parallel computer executes its own program independently. In practice, they execute local copies of the same program. Because the processors do not usually have a common clock either, they execute asynchronously. Hence, they need to synchronize from time to time to be able to cooperate. Since the processors work independently, they need to be able to operate as stand-alone random access machines (RAM). As we noted earlier in Subsection 2.1.1, the mass produced microprocessors are currently the most cost-effective choices for the processing nodes to achieve high per-node performance. Consequently, each of the processing nodes resembles a stand-alone workstation with a substantial amount of memory, communication capabilities, and possibly a local mass storage. Considering the cost of a node, the savings include packaging and the user interface devices, notably the display, which in total form roughly half of the costs. Since the processing nodes make roughly half of the costs of a parallel computer, we can estimate that the cost of a typical MIMD parallel computer is the number of processors mul-

18. Even if the processors are located in the single chip, most of their memory and possibly the communicators are usually located on a different chip. Furthermore, for each multiprocessor chip there may be a separate floating point processor as in the CM-2 computer [100].

tiplied by the cost of a corresponding workstation. The prices of different of mid-cost computers have remained relatively constant over time, while the power of them has increased.[19] Since the typical workstations cost about $5,000-20,000 and supercomputers cost $500,000-30,000,000, we can estimate that MIMD parallel supercomputers have approximately 50-2000 nodes. These estimations are naturally very inaccurate, but they have held well this far, and there are no apparent reasons why they would not be of the correct order also in the near future. A good example of the class of modern MIMD parallel supercomputers is the Cray T3E [25], which includes 32 to 2048 Digital 21164 (Alpha) processors connected together via a three-dimensional torus.

## 2.3 Special purpose computing and special purpose computers

Minimizing the costs of general purpose computers is hard because of the different needs of different applications. If we are designing a computer for a single application, we can choose the optimal solution for every component of the computer. The chosen optimal solution is often much cheaper or better than a general purpose solution. The optimization possibilities include choosing or designing a processor with an appropriate instruction set, including only the minimum amount of memory, choosing suitable memory bandwidth, choosing interconnection network with structure similar to the communication pattern of the algorithm, using precomputed routing to avoid hot spots, and designing a dedicated I/O system from the original data source. The drawback of the special purpose computers is that since they are usually unique, the initial design costs add up totally to the cost of the single computer. An example of a unique special purpose computer is the NAL NWT 2/166 (Numerical Wind Tunnel II) [79]. The NWT 2/166 consists of two control processors and 166 vector processing elements similar to Fujitsu VP400 connected together by a crossbar network.

Because the main focus of this thesis is general purpose computing, we shall not discuss the existing special purpose computers in more depth. One of the goals of our new model is, however, to be able to answer to questions such as “what type of computer should we acquire for this particular application?” or “what feature of this computer should be changed to improve the performance of our application?” The answers to these questions clearly support the design of special purpose parallel computers. The answers are, however, in terms of the new model, and are not necessarily generalizable for different architectures than the shared memory approach. For example, the new model probably cannot guide us to design an efficient hardware based DSP system for image processing applications.

19. We shall not count the top-of-the line workstations since they include, e.g., very expensive graphical display adapters. Furthermore, no MPP computer has yet used the newest processors due to the development delays and the added expenses.

# Chapter 3

# Existing parallel computation models

There are at least two main reasons for computational models. Firstly, any programming effort on physical computers involves many details that are irrelevant for an algorithm designer. As the models ignore the details, the algorithm designer has less things to remember and do. Secondly, the diversity of physical computers is so large that the only possibility to design portable algorithms is to use a higher level model which can be executed by all of the machines.

In this chapter we shall briefly present some existing models and paradigms of parallel computing. In Section 3.1 we shall present the traditional more implicit method of expressing parallelism, the vector operations. In Section 3.2 we shall present the most popular theoretical model of parallel computation, Parallel Random Access Machine (PRAM). In Section 3.3, we shall present some parameterized cost models of parallel computing. Our new model, which we shall present in Chapter 4, also belongs to this class. In Section 3.4 we shall present the message passing paradigm in parallel computing. Finally, in Section 3.5, we shall present the dataflow model of parallelism. The grouping of the existing models is not necessarily generally recognized. We chose the grouping based on the goal of the model and the point of view of the parallelism. For example, the models on the parameterized group all try to characterize the communication capabilities of the parallel machines. Before discussing the models of parallel computations and machines, we present a model-independent classification of parallel algorithms.

### A classification of parallel algorithms

According to Kruskal et. al. [66], the concept of efficiency of algorithms should be independent of the model used, hence they present a classification of parallelism efficiencies. Because the models of parallel computing are very different, the intention of this goal is not very clear. We shall present the classification here because we use some of the terminology later on when presenting the models and sample algorithms. The classification is based on two factors, the efficiency and the parallel running time. The efficiency is defined as the extra work compared to the work done by the best sequential algorithm. The efficiency classes are:

E Efficient, constant inefficiency,
A Almost efficient, polylogarithmic inefficiency, and
S Semi efficient, polynomial inefficiency.

The efficient algorithms are naturally the most desirable. The parallel running time is classified to two classes:

NC polylogarithmic running time, and
P polynomial reduction in running time.

The Nick's Class (NC) aims for very fast parallel algorithms using as many processors as needed. The class P aims for more practical reduction of time, if we have a polynomial number of processors.[20] Note that these classes are not distinct, e.g., many parallel sorting algorithms are in both classes. When pursuing towards the NC algorithms, we have to remember, however, the physical communication latencies in three-dimensional space.

By combining the above classes of parallel algorithms we get six classes of parallel algorithms, ENC, ANC, SNC, EP, AP, and SP. The EP class, efficient, polynomial reductions in execution time, is the most interesting one according to most research. The EP class specifies that we achieve reduction by factor $p$ in execution time using $p$ processors. More accurate complexity classes can be found in [108].

## 3.1 Vector programming model

Scientific computations[21] consist mostly of vector and matrix operations. The operations are performed either independently on every element of the vector, or as a vector operation, such as convolution. In either case, the operations can be accurately expressed as whole vector operations without having to express the iteration over all elements of the vector. In Fortran 90 [78] syntax, instead of the iteration

```
DO I=1,100                                  (3-1)
    X(I) = 2 * X(I)
END DO
```

we can do the doubling with the vector assignment

```
X(1:100) = X(1:100) * 2                     (3-2)
```

or, if we apply doubling to all elements of the array $X$, just

```
X = X * 2                                   (3-3)
```

20. Although the requirement of a polynomial number of processors is not necessarily practical.

21. Such as simulations of behavior of particles or matter according to the laws of physics and chemistry.

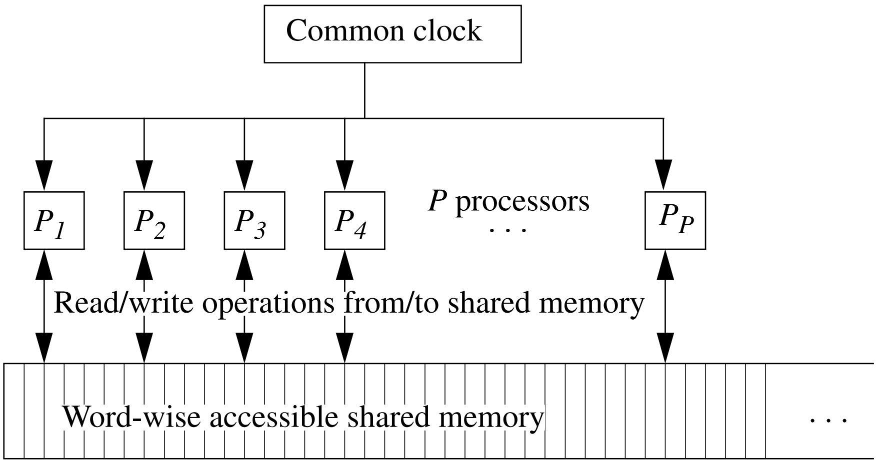


**Figure 3-1:** The structure of the PRAM model.

The vector operations have some advantages over the scalar operations. The most obvious advantages are the easiness of programming and more inherent parallelism. We can easily see that using statements such as (3-3) is much easier and less error-prone than using iterations such as (3-1). Furthermore, the phenomena we are modeling rarely have an iterative nature at the innermost level. Thus, the addition of the iteration is unnatural. On the implementation side, vector operations provide safe parallelization for the compiler. The iterative loops are much harder to parallelize since they may or may not include dependencies between the iterations. Since typical vectors of the inner loops of scientific computations have thousands of elements, they provide enough parallelism for most parallel and vector computers. Currently, and in the near future, the Fortran 90 programming language is probably the most widely used vector programming environment. The APL language [56] is very much based on vector operations. Consequently, it would provide good possibilities for vector based parallelism. Later, we shall use vectors in transferring large blocks of data between local and shared memories.

## 3.2 The PRAM model

Parallel Random Access Machine (PRAM) is a rather straightforward generalization of the Random Access Machine (RAM) model of sequential computation. Instead of one processor and one memory of the RAM, a PRAM has several processors and one memory. Besides this basic fact, the different definitions of the PRAM model vary from one to another. Because of the apparent obviousness of the model, several researchers have independently "invented" the model, and yet more researchers have defined their own versions of the model. One of the first definitions was given by Fortune and Wyllie [34]. Figure 3-1 presents the conceptual structure of the PRAM model. The processors are almost standard RAMs with or without private memory. If the RAMs have private memory, they have also separate instructions for shared and private memory references. Additionally, each of the RAMs has a distinct *Processor-Id* number which distinguishes it from the other processors. The processors execute under a common clock. The choices, whether the processors execute a common program, private identical programs, or different programs, and whether the processors are allowed to branch differently or not, differ according to the definer.

The most important variations of the PRAM model relate to the handling of concurrent shared memory references to a single shared memory location. In most cases concurrent reads and concurrent writes are handled separately, and a read and a write during the same cycle get serialized to a read and a write.[22] Using different policies for read (R) and write (W), we get abbreviations of the form *X*R*Y*W-PRAM, where *X* and *Y* are policies for handling concurrent reads and concurrent writes, respectively. The most used policies are Exclusive (E), no concurrency allowed, Concurrent (C), full concurrency allowed, and Owner (O), only owner of the location allowed. For example, CREW-PRAM stands for concurrent read, exclusive write PRAM. If we allow concurrent writes, we have to define also the method of handling the conflicting writes, or forbid conflicting writes. As with other forbidden operations, in case of forbidden writes, either the results are undefined, or the PRAM halts, again depending on the definer. The most often referenced concurrent write policies are Weak/Common (concurrent writing of zeros / the same value allowed), Tolerant/Collision (nothing / a collision tag is written), Arbitrary/Priority (arbitrary value / the value from the processor with lowest PID is written), and Strong (combination of all writes). The names of the policies may vary depending on the definer. All but the Strong model are rather straightforward generalizations of the CREW model, even if the concurrent write possibility allows us to write unnaturally fast algorithms. For example, a Weak CRCW PRAM can compute the maximum of $N$ elements in $O(1)$ time using $N^2$ processors. The Strong CRCW model is even more powerful, as we shall see in the constant time sorting Algorithm 3-1.

The PRAM model has been criticized of being too theoretical, too strong, and impossible to implement. These arguments are mostly valid, but the PRAM has not been intended for modeling existing parallel computers. Instead, it is intended to be a tool for studying parallelism at a more abstract level. Since the PRAM model is rather strong, and since it can simulate most of the other parallel models efficiently, it can be used for answering questions such as "how many processors we can use efficiently for our task?" or "what is the minimum time required for this task if we are allowed to use as many processors as needed?" The pursuit of the fastest possible algorithms has resulted in even unnaturally fast algorithms. Algorithm 3-1 presents sorting of an array of $N$ elements in unit time by a $N^2$-processor Strong CRCW-PRAM. The algorithm has been used by both the supporters of the PRAM to demonstrate the goodness of the PRAM model, and the opponents of the PRAM to demonstrate the meaninglessness of the model.

Since the PRAM algorithms are abstract and free of implementation-specific details, the PRAM has been the most popular model of parallel algorithm research. Surveys of the algorithms can be found in [57, 62]. Strong support is the biggest advantage of the model since the other models do not have this rather mature algorithmic base [106]. Besides the algorithms, the algorithmic base includes knowledge of lower and upper complexity bounds on groups of algorithms, and knowledge of parallel computability. What it lacks, however, is the knowledge of more accurate efficiency of the execution of the algorithms on real parallel machines, and, naturally, a working example of the PRAM.

---

22. Read before write is probably the more common definition. Some versions, however, use the opposite order.

```
proc sort1(A : array [1..N]) : array [1..N];
    var B, C : array [1..N];
    begin
        for i ∈ 1..N pardo
            B[i] := 1;
        for i ∈ 1..N pardo
            for j ∈ 1..N pardo
                if A[j] < A[i] then
                    B[i] := B[i] + 1;
        for i ∈ 1..N pardo
            C[B[i]] := A[i];
        return C;
    end;
```

**Algorithm 3-1:** The $O(1)$ time sorting algorithm for a $N^2$-processor Strong CRCW-PRAM.

The standard programming approach for the PRAM is the sequential algorithm notation augmented with the *for-pardo* construct to introduce parallelism. The view presented by the programming model is the view of a control processor, which executes the serial code, and launches and controls the parallel executions on the parallel processors. Since the PRAM model provides as many (virtual) processors as needed, the programming model does not determine the way of distributing the subtasks to the physical processors.

Besides being a theoretical model of computation, there has been research to implement PRAM using distributed[23] memory machines (DMMs). In several cases, an $O(dP)$-processor PRAM can be simulated asymptotically efficiently by a $P$-processor DMM with a network of diameter $d$ if the network has capacity $O(dP)$ [87]. The needed extra factor of $d$ processors is called *slackness*. The problem with these simulations is the usually unknown constants and the requirements on the capacity of the interconnection network. Some of the constants of the algorithm can be estimated using simulations, but the constants induced by the hardware are much harder to estimate. Similarly, the cost of the interconnection network is highly dependent on the technology. Thereby, the costs are hard to estimate. In any case, we have to count the cost of the components of the interconnection network, especially if the network has asymptotically more components than the number of processors. For example, a butterfly network uses $P \log P$ queueing and routing capable intermediate nodes for $P$ processors.

## 3.3 Existing parameterized parallel computation models

Since the inaccuracy of the PRAM model has been recognized, and since no generally accepted more accurate parallel computation model has appeared, several researchers have presented their own "more realistic models of parallel computation." In this section we shall describe some of them. In addition to the ones we present here, more or less sim-

23. Or modularized memory machines.

ilar models have been suggested in, e.g., [3, 12, 17, 23, 38, 39, 40, 46, 58, 71, 72, 75, 82, 84]. Surveys and comparisons of the models are made in, e.g., [14, 39, 59, 71, 95]. After we have presented our own model, we shall compare it with the other models in Section 4.7.

### Bulk-Synchronous Parallel (BSP) model

Valiant [104] presented the Bulk-Synchronous Parallel (BSP) model as a new *bridging model* for parallel computation. The term "bridging model" means that instead of being an accurate computation and programming model, the BSP model should be considered an intermediate model between the machines and the programs. The instances of the BSP model consist of a number of processor-memory nodes, the interconnection network, and a synchronization facility that synchronizes the processing nodes at regular intervals of length $L$. Besides being a bridging model, the instances of the BSP model can be considered as parallel programming models. XPRAM model presented by Valiant [103] is an instance of the BSP model. The XPRAM model divides the computation in supersteps of length $c \times L$, where $L$ is *periodicity* of the computer. During each superstep, a processor $i$ executes $a_i$ local operations, $b_i$ sends, and $c_i$ receives. The sends and receives are called global operations. Even if the model refers to messages, it can be used to model the distributed shared memory. The time $r_i$, which is related to the global operations to execute the superstep by processor $i$, is $r_i = a_i/g + b_i + c_i$, where $g \geq 1$ is the cost, in steps per word, of one global operation. The $g$ is calculated as

$$g = \frac{\text{total number of local operations by all processors per second}}{\text{number of words delivered by the communications system per second}}. \quad \text{(3-4)}$$

We shall denote the maximum time over the processors by $t = \max\{r_i | i \in [1, p]\}$. The time to execute the superstep by the whole machine is thus $\lceil t/L \rceil Lg$. After the completion of the superstep the processors are barrier synchronized.

The BSP model has recently gained more support than the other parameterized models. One possible reason for this is that the model does not emphasize the use of the parameter as heavily as some other models. Also, there exists a portable implementation of the BSP model, called BSPlib [49].

### LogP model

Culler et. al. [28] presented the LogP to model current and forthcoming multiprocessor supercomputers. The LogP abstracts the properties of the message passing network to a set of parameters that describe the machine. The parameters of the LogP model are

- $L$ an upper bound on the *latency* of communicating a word,
- $o$ the *overhead* time used in a processor for sending or receiving a message,
- $g$ the *gap* required between successive communications made by the same processor, i.e., the reciprocal of the bandwidth available for single processor, and
- $P$ the number of processor/memory modules.

Furthermore, at most $\lceil L/g \rceil$ messages may be in transit to or from a processor at any given time. The processing nodes and the messages are asynchronous, and the processors synchronize through the messages. The asynchrony of the messages means that the order of the messages is not guaranteed. LogGP model [6] is an extension for the LogP model. The additional parameter is $G$, *gap per byte* for long messages.

### ***Y-PRAM*** model

*Y-PRAM* presented by de la Torre and Kruskal [99] is a recursively decomposable synchronous model of parallel computing. The number of processors in a *Y-PRAM* is $P = 2^p$. Any submachine may block itself from the rest of the machine and operate independently from the rest of the machine. The processors of a submachine execute *periods of computation* and *period of memory accesses*. The memory accesses are allowed only to the memory *owned* by the processors of the submachine. The time required for total $M$ memory accesses within a submachine of size $S$ is

$$\Theta(\delta(S) + m + M\beta(S)/S) \quad , \tag{3-5}$$

where[24]

| | |
|---|---|
| $\delta(S)$ | is the latency of references within submachine of size $S$, |
| $m$ | is the maximum number of references made by any processor, and |
| $\beta(S)$ | is the bandwidth inefficiency within submachine of size $S$. |

As concurrent memory accesses to a single memory location are not allowed, each submachine resembles an EREW PRAM.

## 3.4 Message passing models

Traditionally the opposite of shared memory based parallelism has been message passing based parallelism. Instead of writing and reading the shared memory the processors, or processes, of a message passing system communicate directly by sending and receiving messages. Unless special multi- or broadcast operations are available, the messages are delivered only between two processors, which differs from the shared memory approach, allowing any shared memory location to be read by any processor. The message passing models differ from each other mostly in two ways, whether a processor can directly reach every other processor or only a subset of the other processors, and whether the communicating processors need to synchronize during the communication, or not.

The distributed computing systems also use the message passing approach. The research in distributed systems has, however, been rather apart from the message passing based parallelism research. We shall not present the models used by the researchers of the distributed systems, but we must remember that the boundary is not clear, and that the problems are similar.

24. Informally, $\Theta(f(n))$ stands for a growth rate of *exactly* order of $f(n)$ [63].

The definition of the sender and the receiver of the messages varies depending on the model, and especially on the level of the programming models. Because the nodes of the physical computers have limited degrees, they can communicate only with their neighbors. Direct all-to-all communications are possible only using either the $P^2$-edged complete graph or the bus, which are poorly scalable.[25] In a non-complete network, the messages to the distant nodes must be routed through other nodes. In other words, the routing of a global message has to be decomposed to routing one or more neighbor-to-neighbor messages. A programming model may provide all-to-all connections for the programmer by letting the runtime system take care of the routing of the messages. Using the virtual all-to-all communication, the programmer needs only to specify the processor-id numbers for a virtually direct communication.

The sender of a message naturally needs to specify the receiver of the message, but the receiver does not necessarily need to know the sender of the message. One of the most important features of the message passing models is the receiver's ability to select one of the many potential senders of messages. The selection criteria vary depending on the model, but most models include possibilities of choosing the oldest ready message or a random one of several ready messages. More sophisticated possibilities include prioritizing depending on the sender and even on the contents of the message [9].

The synchrony of communication is the other important factor of the message passing models. Synchronous communication, e.g., the rendezvous model used in Ada [10], requires the sending and the receiving processes to synchronize during the communication. If either of the processes is not ready for the communication, the other has to wait until both are ready. Within a synchronous model, any asynchrony and buffering of messages must be done using additional buffering processes. An asynchronous communication model allows the sender of a message to proceed with its own computation independently of the receiving process. Synchrony between the processes of an asynchronous system can be achieved using acknowledgment messages.

Communicating Sequential Processes (CSP) model [52][26] and the derived occam language [54] use *channels* of communication instead of defining the sender and the receiver. Using channels allows the programmer to write more abstract processes, which do not refer to other processes, but to channels, which the processes receive as their parameters. Using the abstract channels, the CSP processes should appear like pure functions of functional programming.

Message Passing Interface (MPI) [76] and Parallel Virtual Machine (PVM) [37] are the most used implementations for practical portable message passing programming. Both are implemented as a portable set of libraries to be used in C or Fortran source code level to parallelize, or actually to distribute, applications. Both define the library calls for the basic operations such as creating processes,[27] sending and receiving data, and synchronizing the processes. More recently, also the BSPlib [49] library have been used in

25. Using intermediate routing-only nodes, the processors may have virtual all-to-all connections, but the physical structure of the network remains neighbor-to-neighbor.

26. The original, and more referenced, model in [51] did not, however, use the channel concept.

27. The original MPI 1.x version does not allow dynamic process creation, but the new MPI 2.0 version [77] fixes this inconvenience.

place of the older libraries. The libraries present the programmer the all-to-all communication model. In other words, they hide the physical communication network for the sake of portability. In addition to point-to-point messages, the MPI library provides collective communication routines, such as broadcast and gather. All MPI routines allow for sending more than one element of data with one call. In practice, sending larger blocks is vital for the efficiency since the initial costs of the routines are quite large in many implementations.

One of the shortcomings of the MPI and PVM models is that even if they allow us to write portable programs, they do not make possible to portably optimize the performance of the program. As the previously presented parameterized models, these message passing models should include primitives to provide some information of the used machine to the programmer. As we shall see in Chapter 5, it is possible to include these features in the programming model.

We can argue which of the message passing or the shared memory approach is more natural for an average programmer, but it is definitely possible to write efficiently executable message passing algorithms and programs. The problem with these algorithms is that they exploit either all-to-all communication or a specific communication pattern. The algorithms with a specific communication pattern may not be easily portable to other machines with different interprocessor connection architecture. We can simulate different types of communicating graphs by embedding the graph of the algorithms to the graph of the machine. Unless the embedding is very good, the embedding often leads to inefficient use of the resources [69]. Consequently, the programmer may not assume any interconnection topology for a portable message passing program, since wrong assumptions will decrease the performance considerably. For example, an algorithm using a 3D mesh decomposition will be inefficient in a parallel computer using a fat tree topology. Similarly, using standard binary tree algorithms is not efficient if the programs will be executed in mesh-structured computers. Thus, a portable message passing model should support all-to-all connections. Furthermore, we should charge some costs on the using of the connections, which would return us to the parameterized models of Section 3.3.

## 3.5 Dataflow model

Dataflow model of parallel programming is based on the intuition that the maximum amount of parallelism can be extracted from a problem by examining the operations performed on a piece of data. The flow of data constitutes a directed acyclic graph (DAG) of data movements between operations. The DAG directly defines the dependencies between operations. The maximum width of the DAG is the maximum amount of parallelism usable to perform the algorithms, and depth of the DAG is the minimum time required to perform the algorithm if we had parallel enough computer. The input of the algorithm is given as the first, i.e., in-degree zero, nodes, and the output is given as the last, i.e., out-degree zero, nodes. The operation of a node can be performed as soon as all of the inputs are available. After the operation has been performed, the output of the node is available for the consecutive nodes of the DAG.

There have been efforts to build computers which are capable of executing dataflow algorithms [11]. To improve the efficiency of the execution of the dataflow graphs, most of the dataflow computers, and the programming models, have included some exceptions from the pure dataflow approach. The most notable ones are the once-writable I-structures. Usually the dataflow computers are programmed using functional programming languages since they share the same paradigm. On the other hand, dataflow programs can be derived from slightly restricted procedural language [36], but it might be generally too difficult and inefficient.

# Chapter 4

# A new model of parallel computing: the F-PRAM model

As a summary of the previous chapters, we can state that there is a gap between current, and probably forthcoming, parallel computers, and the current, and probably forthcoming, parallel algorithm design and programming conventions. Parallel computers probably cannot have low latency, linear bandwidth,[28] and linear cost random access shared memory. On the other hand, the algorithm designers and application programmers tend to think in terms of data, i.e., shared variables. To fill that gap, we shall present a new[29] model of parallel computing that presents a shared memory model for the programmer, but it should also be realizable on distributed memory message passing computers.

In this chapter we shall define the F-PRAM model and give the rationale for the design choices. To begin with, in the introductionary Section 4.1 we shall discuss the background and the purpose of the model. In Section 4.2 we shall present the components of the model (or, more accurately, the components of the virtual computer modeled by the F-PRAM model). The communication mechanism is probably the most important component in any parallel computing model or system. In Section 4.3 we shall define the most important aspect of the model, namely the cost model, which includes both the computation cost model and the machine building cost model. The rest of the chapter essentially specifies the rationale of the model. In Section 4.4 we shall discuss the rationale of the structure of the model and the number of parameters. In Section 4.5 we shall discuss the methods of designing and analysing efficient and portable F-PRAM algorithms while still trying to keep the design effort reasonable. Additionally, in Section 4.6, we shall discuss the possible optional parameters and restrictions for the model. In Section 4.7 we shall relate the new model with the existing models by giving simulation algorithms by the new model and for the new model. Throughout this chapter we shall also list the rejected features for the model. This discussion must be considered as rationale for the chosen features.

28. Linear with respect to the number of processors.

29. Or, as Vishkin [106] stated, yet another realistic parallel computation model.

## 4.1 Introduction to the F-PRAM model

The models we examined in Section 3.3 are valid proposals to fill the gap, but at the time we designed our new model we felt that none of the models include all the features that are required for a general purpose model. Especially, the existing models are rather abstract, which makes it difficult to examine the programming of the models and the execution possibilities of the algorithms written for the models. The more concrete nature of our new model may restrict its use on some occasions, but allows more accurate analysis of parallel computation. Furthermore, we shall prioritize the features to provide different levels of accuracy depending upon the needs and possibilities in different types of analysis situations. In addition to the model, we shall present a concrete programming model for the computation model in Chapter 5, and an experimental emulator implementation of the model in Chapter 6. Besides the definition of the model, the other important goal of any model is to establish an efficient approach for parallel algorithm design. We shall present the approach, as well as some sample algorithms, in Chapter 7. In addition to the classic analysis of the algorithms, we present the measured performance of implementations of the algorithms. The measurements have been done using a configurable emulator system.

With the new model we shall concentrate on trying to design algorithms that execute nearly $P$ times faster using $P$ processors instead of one processor. Since we do not know the value of $P$ in advance, we should try to make algorithms that can adapt to different values of $P$, i.e., use efficiently as many processors as possible. This resembles the goal of the traditional parallel algorithm research, which often seeks algorithms that can efficiently use asymptotically as many processors as possible to achieve the fastest possible algorithm. Our primary goal is not, however, the maximum asymptotic parallelism with infinite inputs since we have either a parallel machine with a finite number of processors, or a finite amount of money, which can be used to buy a finite number of processors. Furthermore, we do not use infinitely big inputs, but usually as big inputs as we can handle with our machine.

As filling the gap between computers and programmers, it is important to fill the gap between different machines. Filling the gap between to computers enables a possibility to write efficiently portable programs. The new model abstracts the features of parallel computers to a set of parameters that can be used to write programs that adapt to different types of parallel machines. The full set of parameters is rather large, but most of the parameters, called secondary parameters, are only analysis aids and for completeness. In most occasions we need only a few primary parameters for making a program that can adapt to the used computer. The secondary parameters can be used for reality checks to prevent using the possible loopholes left by the primary parameters.

The third important function of the new model is to make it possible to model the importances of the different components of the parallel computers. Using the model we can estimate how the performance of a program will change if we change some features of the computer. We can then answer questions such as "what requirements this particular application has for the parallel machine to be able to be executed efficiently?" or "how efficiently this algorithm can be executed on an Ethernet-connected workstation network?" Besides the model of computation, we should have some information on the relative prices of different components. Then we could answer questions such as "should we buy more processors or a faster interconnection network?", "should we buy wider or faster interconnection network?", "what is the best way to improve the performance of our algorithm?", or "what components we could weaken without decreasing the performance of our algorithm?"

The name of the model, *F-PRAM*, comes from term *Future PRAM*, since the model uses structures similar to the futures of Halstead's Multilisp [42] for shared memory references. Within Multilisp, futures are a more aggressive way of lazy evaluation. Lazy evaluation stands for "evaluation only when needed," and futures stand for "start evaluation immediately at background, i.e., in parallel." In addition to Multilisp, we are aware of two more recent languages that use futures for process creation [20, 21]. As opposed to these languages, the F-PRAM model uses the futures only for shared memory access. For presenting parallelism, we exploit the traditional *par-do* notation and a constant number of processors.

We shall refer to the whole model, including the conceptual structure of the model and the cost model, as the F-PRAM model, or, when there is no possibility of confusion, only as the F-PRAM. On the other hand, we shall call a machine modeled with the F-PRAM model an *instance of F-PRAM*, or, when there is no possibility of confusion, only an F-PRAM.

## 4.2 Components of the F-PRAM model

Components of any parallel computer include processors, memory, a communication facility, and I/O facilities. F-PRAM naturally includes these, and actually little else. There is a limited, machine dependent, number of processor nodes, each of which has some private memory. We shall describe the nodes in more detail in Subsection 4.2.1. Besides the private memories, the F-PRAM model includes some amount of (possibly virtual) shared memory which we shall define in Subsection 4.2.2. The shared memory is the facility that the processors use to communicate with. Processors access the shared memory via an interconnection network that, together with the shared memory, determines the communication facilities of an instance of the F-PRAM model. Throughout this thesis we shall use the general term communication for the processors communicating indirectly by means of accessing the shared memory. The argument for this slight inaccuracy is that the shared memory is the sole medium of communication for the processors. We shall describe the properties of the interconnection network in Subsection 4.2.3. We shall define an important part of the communication, namely synchronization, separately in Subsection 4.2.4. Finally, in Subsection 4.2.5, we shall discuss the I/O possibilities of an F-PRAM. In par with the definition of the components, we shall discuss some background information to justify the chosen features of the components. Figure 4-1 presents the conceptual structure of the F-PRAM model, and the conceptual structure of an instance of the F-PRAM model. Even if the components of the F-PRAM model work together, each of the components has its own view of the model. The view consists of the functionality and the responsibilities of the component itself and a view of the other components that the components communicate with. In the following subsections, in addition to the details of the components, we shall describe the relations of the components to the other components.

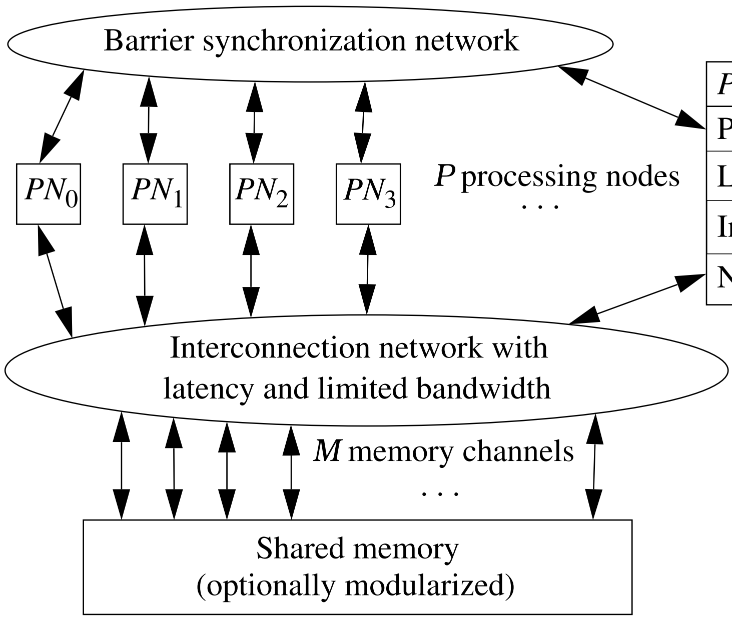


**Figure 4-1:** The structure of the F-PRAM model.

The separation of the different components of the F-PRAM model emphasizes the independence of the operations the components perform. In other words, the components work asynchronously unless explicitly synchronized of they have a need to interact. For example, communication and computation are performed concurrently and independently by the interconnection network and the processing nodes, respectively.

### 4.2.1 A set of processing nodes

The main part of a processing node is an ordinary sequential random access machine (RAM). The RAM has it's own private memory for program and local data. The exact architecture of the processor is not relevant in the model. For example, there will probably be one or more levels of caches between the processor and the local memory, but it does not have on effect on the model. We may use any architecture as the basis of the model since the model does not expect any special processor structures, such as special registers, multithreading, fast context switching, or special fast interrupts. This is because each of the processors works independently sequentially until it needs to interact with another part of the machine. The local memory subsystem must, however, allow the network interface to able to update the private memory. This is not as special as it seems to be. In most current computers there is a special direct memory access (DMA) protocol for the I/O devices, such as disk drive controller, to write the data directly to the main memory.

The number of processing nodes is a constant, machine-dependent number. Thus, we can refer to it as a regular variable without, for example, necessity to use any functions of the size of the input. The important advantage of this simplification is that also the other measures that depend on the number of processors can be referenced as regular variables that can be evaluated on compilation or load time.

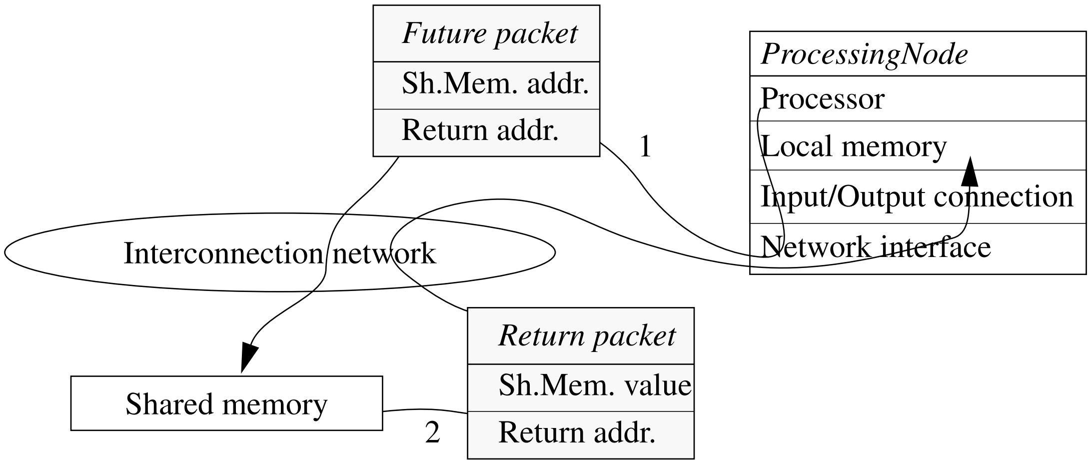


**Figure 4-2:** A route of a future request.

### Shared memory references

In addition to the normal RAM operations, the processor is able to issue shared memory references to the network interface. Here a shared reference stands for either a request to write a local value to a shared memory location or a request to obtain a value of shared memory location to the local memory. The processor delegates a shared memory write to the network interface, that delivers it through the interconnection network to the shared memory. From the viewpoint of the processor, the write only takes a constant time. A shared memory read results in a *future*. A future is a temporarily tagged piece[30] of local memory. The tag tells us whether the future has been resolved or not. In practice, each word of memory can contain either a normal value or the tag as a future marker.

Issuing a future takes a constant amount of processor time. While a future is being resolved (i.e., served) by the shared memory and interconnection network, the processor is able to continue its execution. It also may issue more futures. The value of an issued future is unusable until the future has been resolved. The processor can check the state of a future by inspecting the tag of the future. Figure 4-2 presents the route of one future request from the processor to the network interface, and via the network to the shared memory, and back to the local memory. We shall present the functions of the shared memory and the interconnection network in the next two subsections.

When the network interface receives the value of a previously issued future, it places the value to the memory location of the future and clears the tag to show that the value has arrived. The processor is not disturbed by message receiving. The next time the processor inspects the future, it gets the value. There is no need for any acknowledgment signals from the network interface to the processor. If the processor has nothing useful to do, i.e., it needs the value from the shared memory, then it needs to wait for the future to be resolved. Because the waiting is not useful work for a processor, the F-PRAM model advocates that the algorithm provides useful work for the processor to do before it tries to read the value of the future. This way we can ensure that no useless waiting occurs.

30. Implemented, for example, as a special not-yet-available value within a single word, or as either two words of memory, one of which is the tag, the other is the data.

As opposed to the current SMP computers, the F-PRAM processing nodes do not cache the shared memory. The future protocol makes the caching unnecessary. If a process needs a value from the shared memory, it requests it beforehand using a future. Thereafter, the value is kept in the local memory as long as the process changes it or rejects it. Consequently, the local memory works as a cache of the shared memory, except that the shared memory values are not updated automatically.

### 4.2.2 Shared memory

The shared memory is probably the most difficult component in parallel machines to model well while maintaining a reasonable simplicity. The most straightforward and powerful definition would be that we have an $M$-port monolithic shared memory that can handle $M$ nonconflicting memory requests simultaneously, i.e., on every clock cycle. Nonconflicting means that there may not be two writes, nor a write and a read, to the same memory location during the same clock cycle. The "one write and one or more reads" case could be settled by defining that all writes take place before all reads, or vice versa. This, however, might take twice the time than the plain nonconflicting solution. Implementing a true multiport memory with independent access possibilities to every memory cell in spite of all other references is probably rather expensive [32]. There should be an independent path from every port to every memory cell, which would result quadratic costs in the VLSI area.

A more realistic shared memory model is the model of $M$ memory modules, each capable of handling one request at every memory cycle. The memory locations of the logical monolithic shared memory are distributed among the $M$ memory modules. We shall assume that each of the shared memory locations exists as a single copy, i.e., we use no redundant memory. The complex thing in distributing the memory locations to several modules is choosing the method of distribution. Some popular ones are the randomized, block and cyclic distributions. With any given distribution, it may happen that on some occasions several concurrent references have the same destination. Using a good randomized distribution, it is possible to prove that out of $P$ random requests to $P$ modules, on average $O(\log_2 P)$ requests will be addressed to a single module with high probability. Using other distributions, such as cyclic or block, there can be occasions when all requests are destined to a single module, which totally serializes the requests. Those types of problems can be solved by allowing the programmer to guide the data distribution to avoid the bad reference sequences in the program. This has been proposed, albeit partly for slightly different reasons, in the High Performance Fortran (HPF) standard [47]. The approach has the limitation of being mostly suitable for vector operations and making the programmer's work more complex. We shall discuss the possible distribution directives for the programming model of the F-PRAM model in Section 5.3.

The basic paradigm of the whole F-PRAM model is to have a rather plain basic model and a set of possible refinements for more accurate inspections. Consequently, the basic conceptual shared memory model of the F-PRAM is the monolithic $M$-port memory system that can serve at most $M$ nonconflicting memory requests on each clock cycle. The attribute "at most" means that, unless we include additional restrictions in the model, we can assume the $M$ requests to be served on each cycle. The restrictions can include restricting simultaneous reads from a single location and simultaneous references to a single memory module. Furthermore, the latter refinement requires us to specify the data distribution method as well. A further possible restriction would be the size of the shared memory, but it is probably useless with most parallel applications and with most parallel computers. We shall discuss the costs of using modularized shared memory model in more depth in Subsection 4.3.2. As a general assumption to achieve enough bandwidth, the machines should have at least $P$ memory modules. To compensate the impact of slow memory and to reduce hot spots, we should have $M > P$ in a high performance machine. Vector supercomputers have $M{:}P$ ratios in the range 32-256 [17].

If we assume that the memory system is constructed of $M$ modules that handle at most one request on every cycle, the requirement of no conflicting references is fulfilled automatically. All requests to a single memory location will be serialized since the module handles the requests one by one. The fact that the requests might have been issued simultaneously does not affect the functionality of the memory module since the interconnection network and the network–memory module connection will serialize the requests. An advantage of the F-PRAM model is that the order of the handling of the concurrent references does not matter.

### 4.2.3 Communication network with latency and bi-section bandwidth

The task of the interconnection network is to deliver shared memory reference packets from the processing nodes to the shared memory modules and back. The interconnection network defines the minimum time needed for a shared memory reference. In addition to bi-directional communication, also the time needed for the shared memory to respond to the arrived request must be taken into account. The access times of the memory modules can be considerably longer than the clock cycles of the processors, but is often low compared to the interconnection network latency.[31]

We briefly discussed the properties of some possible network topologies in Subsection 2.1.3, but in a model that tries to cover all parallel computers, we cannot choose the topology of the network. Instead, we have to select a set of features that characterize the properties of the network accurately enough. It is natural that the faster the network delivers the requests from processor to the shared memory and back to the processor, the better. In other words, the *latency* of the network should be small. Another important feature of a network, perhaps even more important, is the capability to deliver as many messages as possible between arbitrary nodes of the network. Because a big part of the packets are routed across the network, the network should have a high *bi-section bandwidth*. Furthermore, a good network has a big *capacity* to be able to hold as many messages simultaneously as possible. The reason for the capacity is to simplify the structure of the processing nodes. If the network can hold additional messages, the processing nodes need not to queue the messages.

The latency of a network has no generally accepted definition. A plain definition would be to measure the time needed to deliver a message, but the problem is that the times vary from message to message. Proposed definitions include minimum time, average time, average worst time, and absolutely worst possible time. Additionally we can measure the times either on empty, partially loaded, fully loaded, or overloaded network.

31. In SMP machines, however, the memory latency may surpass the network latency.

The information of minimum possible time on an empty network is of little use, since it is a practically nonexisting phenomenon in practical computations. The average time is a better measure, but not all messages are delivered within the average time. Especially the slowest messages never get delivered within the average time. Since all messages probably are important, we would not have any knowledge of how long we have to wait to receive all messages. The average worst time tells us the expected time after which all messages have been delivered. This is usually the most important latency measure we need to know when designing parallel algorithms. The absolute possible worst time is hopefully rare enough to be neglected in modeling.

When we discuss a read request across the network, the latency has to be multiplied by two. In addition to the packet delivery time, the latency appearing for the processor can include some packet constructing, queueing, and acknowledgment times. The impact of the load of the network on the latency is rather hard to model properly. A good network with a good routing protocol should be as insensitive as possible to the changes in the load of the network. In other words, the latency should remain constant, or grow only moderately, until the network really is overloaded. A good routing algorithm should also be able to handle overloads with sustained throughput.

The bi-section width of a graph is defined as the minimum number of edges that have to be removed to split the graph in two halves of an equal number of nodes. The bi-section bandwidth of an interconnection network is often defined as bytes/second across the bi-section of the network. In a parallel computation model, a better measure is words/processor/instruction, i.e., the number of words each processor can send on each clock cycle without saturating the network. Since we do not use nearness concepts, all the packets have to be assumed to be global. Because of the globality, this measure combines both the bandwidth and the capacity measures of the network. In a good machine the words/processor/instruction should be a small constant, in practice at most one, unless we use shared memory-to-memory operations, which would require two. A less efficient machine would have a bandwidth that follows a function of the number of processors. Since the measure is usually less than or equal to 1, we shall use the reciprocal of the words/processor/instruction measure, called *communication inefficiency*. It determines how often, in clock cycles, a processor can communicate without saturating the network. The network saturation is not likely to happen with a very small number of successive messages, e.g., two references in successive cycles should not saturate the network whatever is the communication inefficiency. Thus, we shall assume that the inefficiency restriction is measured as an average load within a longer period. A reasonably long time would probably be the latency time of the network.

In a normal situation, the capacity of the network should be the number of processors multiplied by the latency divided by the communication inefficiency. This shows that communication inefficiency actually abstracts bandwidth and capacity to a single measure. Furthermore, the measure is processor-oriented rather than network-oriented. This is an advantage because the source of the messages, i.e., the processors, should be able to regulate the frequency of the messages.

Single standard messages in the F-PRAM are short, consisting of only two words, as we shall see in Section 4.4. The length of the messages depends on the ratio of the word length to the width of the edges of the interconnection network. In any case, however, the length is a rather small constant. Hence, using any of the popular advanced protocols, such as wormhole routing, makes little difference in the F-PRAM interconnection network, and we shall not expect any specific protocol. The longer messages of the block references can be handled via constant length packets.

### 4.2.4 Synchronization medium

Since the F-PRAM processing nodes work independently without any global control, they need some synchronization mechanism to be able to work together on the same task. Since the interprocessor communication is provided via shared memory references, the communication does not ensure synchrony as the message passing mechanism does. On the contrary, the shared memory communication method requires separate synchronization to ensure correct data is delivered. A read before a delayed write would cause an old or undefined value to be transmitted instead of the correct one. The synchronization can be done using purely shared memory references via locks and/or counters. These, however, are quite inefficient methods. Therefore, we shall define the synchronization facilities separately from the shared memory facilities.

In Subsection 2.1.3 we discussed the difficulties of defining an arbitrary synchronization set. To keep the model reasonably uncomplicated, the only synchronization primitive of the F-PRAM model is thus the barrier synchronization of all processors. In the global synchronization all processors are supposed to execute a synchronization instruction, or a procedure call, which is completed after all processors have called it. If some of the processors fail to execute the synchronization instruction, the other processors have to wait forever. Consequently, the programmer and the compiler have the responsibility to make sure that all processors execute the synchronization. The delay between the moment when the last processor executes the instruction, and the moment when the processors proceed with the computation is called *synchronization delay.* It is a machine-dependent measure, and usually a sublinear[32] function of the number of processors. Because of an unbalanced algorithm, part of the processors may naturally encounter additional delays before the synchronization. These delays, however, have nothing to do with the machine-dependent synchronization delay.

Even if the processors formally synchronize the phases of their instruction streams, the main reason of the synchronization is the synchronization of communications, i.e., the shared memory references. The natural interpretation of the synchronization is that it will ensure the order of the references of different phases. To put it exactly, all references to a single memory location that are issued before the synchronization, will be completed before any of the references that are issued to the single memory location after the synchronization. Otherwise the synchronization would be meaningless. The definition that ensures the correctness on the level of each memory location is the minimum that is needed, but it may be hard to implement efficiently, especially on computers having a single stage network and a dedicated synchronization network. For simplicity we have to assume that all shared memory references that are issued before the synchronization, will be completed before any of the references that are issued after the synchronization. This implies that the synchronization takes at least as long as a shared memory reference, but

32. If we have a bus-based interconnection network without broadcasting capability, the synchronization time is linear with respect to $P$.

this is not a severe restriction since the processors synchronize for the sake of communication.

Even if we define the synchronization network as a separate entity in the F-PRAM model, we must note, however, that the synchronization primitive can be implemented using the interconnection network that is mainly devoted to the connections of the shared memory. It can be even implemented using the shared memory itself. The only aspect we need to know about is the time required to synchronize the processors.

In addition to the whole machine synchronization primitive, the compiler, or the programmer may provide some submachine synchronization possibilities. For example, we can synchronize the processors using a counter when accessing a certain variable. Alternatively, we can use the Algorithm 4-1 (page 80) for synchronous message passing between two or more processors. For synchronizing the execution of a small set of processors on a computer with high synchronization cost, we can use the Algorithm 7-9 (page 160).

### 4.2.5 Input/output facilities

Most of the existing parallel computing models totally ignore the I/O facilities of parallel computers. Similarly, some parallel computers have inadequate I/O facilities for some problems involving big data sets. In this subsection we shall model the I/O facilities of the F-PRAM model. We must note, however, that the I/O facilities are not the most important part of the model. We include the I/O facilities to the model mostly for completeness. For example, the parameter concerning the I/O speed can be used to check that the I/O possibilities of the parallel computer are adequate for the given purpose. Improving the I/O can sometimes be the easiest way to improve the performance of a system in a particular application.

The parallel I/O is not an uncomplicated concept. The $P$ parallel I/O lines coming from the $P$ parallel processors have $P$ other ends that have to be connected somewhere. We can think the I/O consists of several very different levels. It is rather easy to implement parallel disk I/O by having a disk in every node [2], but a massively parallel Internet connection will certainly get narrower somewhere not too far away from the machine. We must remember the source of the data and the destination of the results to be able to decide the boundaries of the I/O system. If we consider a single batch job, for example, a crash test simulation, the data comes to the computer via a network, or on a computer tape, serially, is then computed, and finally returned using the same medium. If the volume of the data is big compared to the needed computation, or the I/O connection slow, the I/O will be a bottleneck rather easily unless the data can be kept on local storage for longer times. If we consider, e.g., a large database, the biggest amount of data is rather permanently within the computer, probably distributed among the local disks[33] or memory. Because the request is usually rather small, it usually takes insignificant I/O time. The result can be, however, rather large.

A system that is dedicated to a single task always gets its input from the same source, and saves the data to the same destination. For such a system we can build a ded-

33. If the disks are central, the I/O bandwidth between the database data and the processors will probably be a problem.

icated I/O from the source and to the destination to ensure a smooth continuous operation. A general purpose system, however, has to have a fast enough general purpose I/O to be able to gather the required data. The problem with the parallel general purpose I/O is that few sources can do parallel I/O. Consequently, a parallel I/O system should be used to parallelize several I/O streams from different sources and to different destinations. Additionally we should be able to perform other computations concurrent with the I/O.

Since the I/O is such a diverse subject, we only include the speed of loading/saving data to/from the machine as a parameter of the F-PRAM model. This can be used mostly to detect bottlenecks of parallel computers or parallel computations. Each of the nodes of the F-PRAM model has its own I/O facilities. The input, or the result, does not have to go through the shared memory since the I/O facilities are a part of the processing node. This is a difference between the formal PRAM model and the F-PRAM model. Most PRAM algorithms presume that the input of the algorithm is located in the shared memory when the computation begins, and the output of the algorithm is left to the shared memory after the computation terminates. We can argue whether this is a reasonable assumption or not. Definitely, there has to be some I/O system if we want to do some useful computation. A sequential I/O system to fill and exploit the contents of the shared memory would add an $O(N)$ I/O time for every algorithm. The linear addition destroys the efficiency of many parallel algorithms. On the other hand, if we want to research only parallel algorithms without any I/O, or other real world aspects, then ignoring I/O of the shared memory is a convenient abstraction. Within the F-PRAM model we could not ignore the I/O because we implemented the model experimentally, and the experimental implementation needed some type of I/O.

## 4.3 Cost models

The main purpose of the F-PRAM model is to model the costs of executing algorithms on parallel computers. We shall consider the costs from two viewpoints: the costs of building a parallel machine and the costs of executing algorithms on a given machine. By combining these two viewpoints, we can also estimate the costs of executing an algorithm on a given machine. Since the technologies used in different parallel computers vary so much and change so quickly, we cannot model exactly, say in dollars, the costs of building a machine according to a given set of specifications. Consequently, we shall leave the counting of the dollars to future research. Instead, we try to keep in our mind things such as "doubling the number of processing nodes doubles the costs" or "increasing the number of wires in the communication network will cost some amount of money."

To begin with, we have to model the machine with the parameters of the F-PRAM model. As the result of that analysis, we have a cost for each operation that is used in an algorithm. Using the costs we can count the time needed to execute the algorithm using rather normal algorithm analysis methods. Furthermore, the used machine may pose some restrictions that have to be verified with the analysis. The difference from the traditional parallel algorithm analysis is that we charge different costs for different operations and on different machines. Thus, expressions of the form "optimal algorithm" change to form "optimal algorithm on the machines that have such and such properties."

**Table 4-1:** The parameters of the F-PRAM model.

| **Primary parameters** | | **Typical/default values** |
|---|---|---|
| $P$ | The number of processors | |
| $L$ | The latency of shared memory references | $L_0 + 2\times\text{Ø}^{a}$ |
| $B$ | The bandwidth inefficiency | $O(1)$ ... $O(P)$ |
| **Secondary parameters** | | |
| $B_P$ | The shared memory reference overhead | $O(1)$ |
| $B_B$ | The block reference bandwidth inefficiency | $\leq B$ |
| $B_V$ | The single variable bandwidth | 1 |
| $B_M$ | The single memory module bandwidth | 1 |
| $M$ | The number of shared memory modules | $P$ |
| $B_{IO}$ | The input/output bandwidth inefficiency | 1 |
| $S$ | The synchronization delay | $\geq L$ |

[a] the diameter of the network

The execution cost model, i.e., the parameters, have three purposes. Firstly, they help us to write programs that tolerate imperfect properties of the parallel machine, i.e., programs that are more portable. Secondly, they help us to determine which types of parallel computers are best for the given algorithm, and vice versa. Thirdly, given an algorithm and a parallel computer, we can estimate the running time of the algorithm more accurately than with more a general model.

In the first two subsections of this section we shall define the parameters of the F-PRAM model that define the behavior of algorithms and the costs of executing the algorithms. Different parameters have slightly different roles in the model. Some of the parameters simply define the time needed to accomplish an operation on a given machine. The rest of the parameters are of a restrictive nature, e.g., they define how many operations we can perform within a given time on a given machine. We shall divide the parameters to two classes, *primary* and *secondary parameters*. Primary parameters model the most important features, and have to be considered always when writing or analysing an algorithm or a program. We shall define the primary parameters in Subsection 4.3.1. The secondary parameters exist for checking afterwards that the written program is executable and efficient on a given machine. We shall define the secondary parameters in Subsection 4.3.2. Table 4-1 presents a summary of the parameters with some approximations of typical values of the parameters. In the first definition of the model [61] the parameter synchronization delay was included in the primary parameters. In Chapter 8 we shall estimate the actual values of the parameters for some real parallel computers.

Ultimately, the execution cost of a given algorithm is the product of the time needed to execute the algorithm and the cost of the machine divided by the lifetime of the machine. The problem of this approach is that since the technologies vary and develop so much, we cannot model exactly, say in dollars, the costs of building a machine according a given set of specifications. Therefore, we have to settle on comparing relative costs of different features. We shall discuss such costs in Subsection 4.3.3.

### Choice of the unit of the parameters

Most of the parameters characterize the time needed for an operation in a parallel computer. In a model of computation the obvious candidates for time unit are “step,” “operation,” “clock cycle,” and “second.” The purpose of the unit is to be able to compare and add different costs induced by different operations. In other words, we should be able to express everything in the chosen time unit. This rules out most of the candidates. We cannot tell that “as the computation takes $X$ steps and memory reference $Y$ steps, the whole thing takes $\max(X, Y)$ steps” since the steps of the computation and the communication are not identical nor comparable. Similarly placing “operations,” “seconds,” or “nanoseconds” into the previous sentence would sound less than perfect. Consequently we have to settle for “clock cycle” since we can at least approximate (or measure/count) the number of clock cycles needed for an operation. Furthermore, we have implemented an emulator system to test the model, and within the emulator the clock cycles are very natural units compared to, e.g., seconds.

We shall define most of the cost parameters as average costs since defining the costs of a single operation accurately in every situation would be far too difficult and lead to far too complex set of cost parameters. Besides average over successive operations, the average is also taken over different processors. This is because we shall assume that all processors are executing the same task. Hence the processors probably execute similar operations simultaneously. We are usually interested in the cost that is likely to be needed to complete all computations. Hence the average cost parameters do not mean literally the average cost of the operations, but the average expected worst case cost over the processors.

Since we measure the properties in clock cycles, we have to assume that an average local operation can be performed in one clock cycle to be able to compare the costs. The problem in the use of a clock cycle as a measure is that even if we are given the costs of some parallel operations in clock cycles, we do not know the costs of the basic local operations, such as an assignment or evaluating an expression, in clock cycles accurately. The costs are machine dependent, and to be accurate, we would have to have parameters also on the costs of the local RAM operations. The structure of those would be very architecture dependent, e.g., depending on the local memory bandwidth. To hide the units from the programmer, we shall provide programming model primitives to determine the costs of sequential basic blocks of our algorithms. These values are counted by the compiler and provided as read-only variables.

Since we are describing a model of computation, not a real parallel computer, we have the liberty to abstract the local operations as well as the parallel operations. For example, if a constant length body of an iteration takes $a$ time, and one synchronization takes $b \times \log_2 P$ time then $\frac{b}{a}\log_2 P$ iterations of the loop take as long as one synchronization. For simplicity we shall sometimes use the term “operation” and the $O$-notation, which ignores all the constants. Consequently, we shall state that the body of the iteration takes $O(1)$ time and the synchronization takes $O(\log_2 P)$ time. Furthermore, we shall state that $O(\log_2 P)$ iterations of the previous loop take as long as one synchronization, even if it is not very accurate. We must note, however, that for slowly increasing functions, such as the $\log_2 P$, the $O$-notation may hide constants that are larger than any practical value of

$\log_2 P$. Hence, we shall be more careful with constants when using $O$-notation with sublinear functions.

If we want to estimate the execution costs with constants instead of only $O$-notation, we have to have some approximations on the costs of the local operations. The simpler these approximations are, the more feasible is the analysis of the algorithms. We can, for example, assume that each assignment, arithmetic operation, comparison, and so on, take one cycle, and adjust the parallelism structure costs to this measure. In any case, to be fair with the constants, we have to count all operations, not, for example, only comparisons of a sorting algorithm, or only multiplications of a matrix multiplication. In practical algorithm analysis this results using one or more constants along the parameters and input size in the time complexity expressions. Accurate estimations can be done only using the real F-PRAM compiler and the emulator, which we shall present in Chapter 6.

### 4.3.1 Definitions of the primary parameters

The primary parameters characterize the most important parallelism features of a given parallel machine. They are presented as machine-dependent variables that are available for the programmer, and are assigned with the machine-dependent values on compile or loading time. All of the parameters have two purposes. Firstly, we can use them in the algorithm to write a program that behaves according to the features of the target machine. Secondly, the parameters tell us the costs that are used in the analysis of the algorithms.

#### The number or processing nodes $P$

Most existing parallel computation models allow us to use as many processors as we need. The number of processors that an algorithm designer uses is usually a function of the size of the input. When executing such an algorithm, either the compiler, or the runtime system have to map the requested virtual processors to the real processors. Even if we still present this same abstraction for the programmer in F-PRAM, while executing or analysing an algorithm, we have only a limited number of processors available.

**Definition 4.1:** The parameter $P$ stands for the *number of processors* the algorithm can use during the computation. Each of the processors has a unique number, *processor-id* (*PID*), ranging from $P_0$ to $P_{P-1}$.

The "availability of $P$ processors" means that we can use a parameter $P$ and a set of $P$ processors named from $P_0$ to $P_{P-1}$ to execute the algorithm. The value of $P$ is not available for us until we fix a machine with the algorithm. When writing an algorithm, we can still present more virtual parallelism than the parameter $P$ would allow. The compiler should then be able to map the work to the $P$ processor, but it will not create any virtual processors. Particularly, the compiler will not apply for any more physical processors even if the algorithm used more than $P$ parallel threads of execution. Consequently, when analysing an algorithm, we have to express the parallelism in terms speedup $X$ with $P$ processors. We shall discuss the practices of analysis in more depth in Section 4.5, and the distribution of the work among the processors in Chapters 5 and 6 with our programming model and compiler for the F-PRAM model.

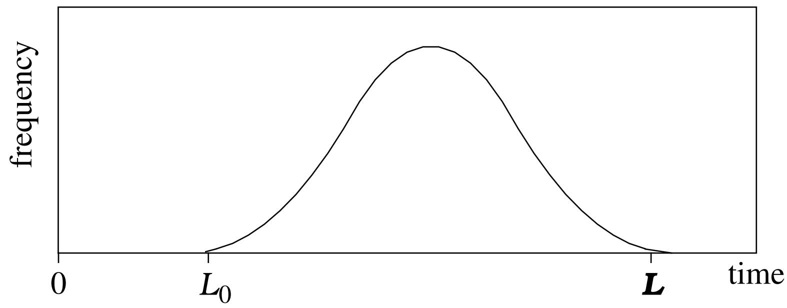


**Figure 4-3:** A possible latency distribution.

### Latency $\boldsymbol{L}$ of shared memory references

After a processing node has issued a future or a shared memory write request, it continues its execution immediately at the next instructions. The impact of the request, i.e., the write or the resolving of the future, will happen independently after some time. To fill that time we should write algorithms that allocate some useful work for the processor before it will try to use the value of the future. Since there is no signaling from the network interface or the memory to the processor when the result of the future arrives, we need to know the approximate time when the result is expected to arrive. More accurately, we are interested in the time within which all the futures have arrived with high probability.

**Definition 4.2:** The parameter $L$, the *latency of shared memory references*, stands for the expected worst time needed for a shared memory reference to be completed.

Figure 4-3 presents a possible distribution of the latencies of the shared memory reference and the proper value for $L$ within the distribution. As with the parameter $P$, also the parameter $L$ is available for us as a variable until we fix a machine with the algorithm, when it becomes a constant. The definition of latency is useful mostly in future requests, since in case of shared memory write requests, the processor that issued the reference probably is not interested in reading the new value and other processors have no knowledge at the moment of the issue. Furthermore, since the processors are asynchronous, there should always be a synchronization between a write to a memory location and a read from the same location. Hence, the latency of a shared memory write is rarely used, and we define it only for completeness.

A typical use of the parameter $L$ in an algorithm is to issue $L$ futures before starting to use their values. If we do some other operations in addition to issuing the futures within the same loop, we need to iterate the loop only $L/x$ times, where $x$ is the number operations within the iteration. The value of $x$ in clock cycles has to be provided by the compiler as the programmer probably has difficulties in determining it. More easily, we can choose between a fast routine and a latency tolerating routine according to $L$. An example of this approach is presented in parallel compare-exchange in Algorithm 7-3 (page 125).

Figure 4-4 presents a timing diagram of the communication (diagonal arrows) between the participating components (horizontal arrows) of the future requests. The processor does local computation between the future references and only later uses the value it requested to be transferred from the shared memory to the local memory. The task of

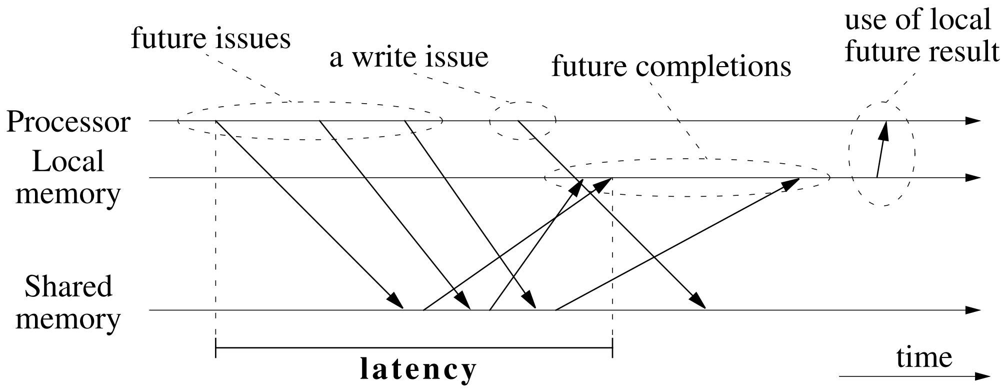


**Figure 4-4:** A diagram of the latency of the shared memory references.

the interconnection network is to deliver the references between the processing nodes and the shared memory. We shall use similar timing diagrams throughout this chapter when defining the parameters.

The F-PRAM model does not require the futures or the shared memory writes to complete in the same order they were issued. Only a synchronization ensures the order of references. This asynchrony is quite different from some other models. The reason for this relaxation is that few algorithms actually need the sequential consistency of the shared memory references. We believe that this relaxation could simplify the implementation of the processing nodes and the shared memory. For example, if we use randomized hashing of the shared memory locations and packet-based network protocol, maintaining the order of the references would require extra work.

### Bandwidth inefficiency *B*

Few parallel computers have such a good communication network that could deliver all the messages the processors are able to issue without saturating the network. Consequently, we have to know how many references we can issue within a given time without saturating the interconnection network. On the other hand, when analysing an algorithm, we need to know how long it takes to transfer a given volume of data. Since the processors work independently and the time is measured in clock cycles, a natural measure would be references/clock cycle/processor. Because this measure would be less than or equal to one in most cases, we shall use the reciprocal of the previous measure.

**Definition 4.3:** The parameter $B$, the *bandwidth inefficiency*, stands for the reciprocal of the maximum shared memory referencing capacity available for each processor, in cycles/reference.

In other words, $B$ is the minimum number of clock cycles between successive shared memory references issued by a processor to ensure that the interconnection network is not saturated. The restriction of the minimum number of clock cycles between successive shared memory references is not strict, but should be understood only as a guideline. More informally, we define $B$ as the minimum average number of clock cycles between successive shared memory references over time $L$ on an *average* processor to avoid saturating the interconnection network. More accurate restrictions are defined with secondary

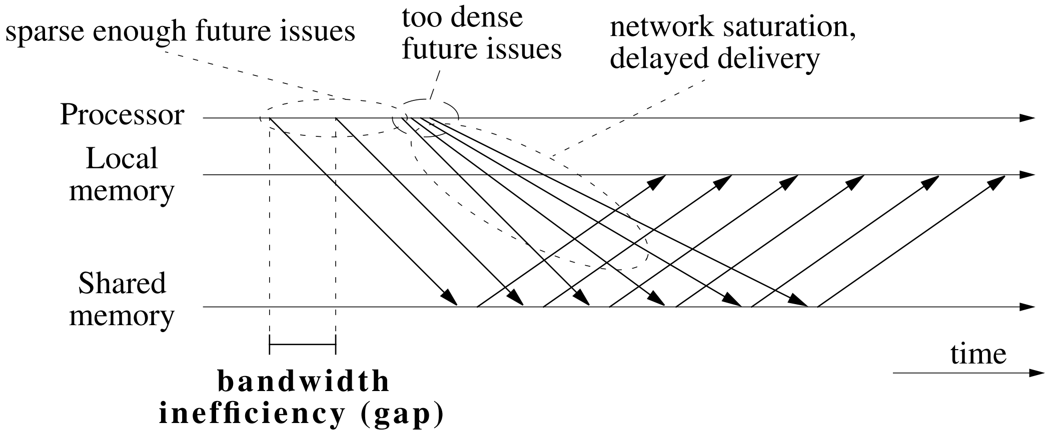


**Figure 4-5:** A diagram of the impact of bandwidth inefficiency.

parameters in the next subsection. These allow us to check restrictions with each processor beyond the accuracy of the averages over time and processors.

Figure 4-5 presents an example timing diagram of the impact of bandwidth inefficiency. If a processors issue futures too often, the network saturates, and the delivery of the later futures get delayed. The processors, however, do not notice the delay unless they try to use the results of the futures. The figure also reveals that our parameter $B$ resembles the parameter gap ($g$) of the LogP model [28]. Instead of using the existing term, we use the "bandwidth inefficiency" to emphasize that $B$ slows down the communication but does not force the processors to wait between communications.

The above definition referring to an average processor includes a possible loophole for situations where only part of the processors communicate. Using a naive average, we could state that each of the $p < P$ communicating processors would apparently have bandwidth inefficiency

$$B' = B \times \frac{P}{p}, \tag{4-1}$$

which would naturally be unrealistic if $B \neq O(1/P)$ and $p \ll P$. On the other hand, the formula is valid, e.g., for bus-based communication networks. The secondary parameter $B_P$ does restrict the bandwidth of a processor, but cannot take into account the hot spots of the interconnection network. Consequently, we cannot guarantee the average-derived bandwidth of Formula (4-1) for unrealistic values.

A typical use of the parameter $B$ is to compare the cost of the volumes of the communication with the cost of the computation of a given algorithm. If the bandwidth inefficiency of the parallel computer is large and if the algorithm communicates much more than it computes, it might use asymptotically more time communicating than computing, which is not efficient. An extreme example of a high-$B$ parallel computer is a network of workstations, NOW for short. A typical NOW might have 15 workstations of 50 MIPS each connected with only a 10 Mb/sec Ethernet network. Such a system would have $B$ values around tens of thousands. When using such a system we have to be very careful to make sure that the communication volume is considerably smaller than the amount of computation.

The above definition "without saturating the interconnection network" is not very accurate. Alternatively we could define an option to force the restriction on the communication volume. We could define that a shared memory reference takes max(1, $B-Y$) time,[34] where $Y$ is the time used by the operations performed after the previous shared memory reference. The disadvantage of this definition is that it prevents nonuniform use of the bandwidth. Consequently, we shall leave also this aspect of the bandwidth open, but since this definition provides a solid worst case method, we shall use it later in some analyses.

### 4.3.2 Secondary parameters

The role of the following secondary parameters is to enable a more refined analysis of our algorithms. For example, we can use the secondary parameters to ensure that our algorithm is efficiently executable on a given machine even if we take the I/O into account. Alternatively, we can attach some requirements on the features of the computer to the algorithms. Furthermore, we can check, how much an algorithm would speed up if we would use some strong feature of a special parallel computer architecture. The other important task of the secondary parameters is to fill the possible loopholes left by the primary parameters. In other words to prevent using any "smart tricks" that improve the theoretical performance of the algorithm, but are not realistic on real parallel computers.

We do not bind the set of the secondary parameters tightly. We can later add more features to be modeled with the secondary parameters if needed. We must remember, however, that the algorithms that use some new exotic features are less portable and harder to analyse that the algorithms written with only the basic features. Therefore, we should design the basic algorithms without the use of the secondary parameters. Implicitly, the design without the secondary parameters corresponds to the use of the secondary parameters with their default values. Unlike the primary ones, the secondary parameters have rather apparent default values.

#### Shared memory reference overhead $B_P$ of the processors

The processors of some parallel computers do not have as good network interface coprocessors as we assume within the basic F-PRAM model. In such a system the processors have to participate in communication. In practice, at least constructing the future packet and injecting it to the network might be a responsibility of the processors. Also receiving a message would possibly raise an interrupt for the processor to execute the message receiving handler. Besides receiving its own messages, a processor might have to serve other processors's requests for the part of the shared memory it maintains. If the processors are located at the inner nodes of the interconnection network, they may have to participate in routing of the messages in the worst case.

Since there are several factors affecting the processor time needed in different phases of future references, we shall not try to model them separately. Instead we use an estimated average cost induced by a communication operation. Since the total processor

34. More accurately max($B_P$, $B-Y$) time, but we shall define the parameter $B_P$ later in Subsection 4.3.2.

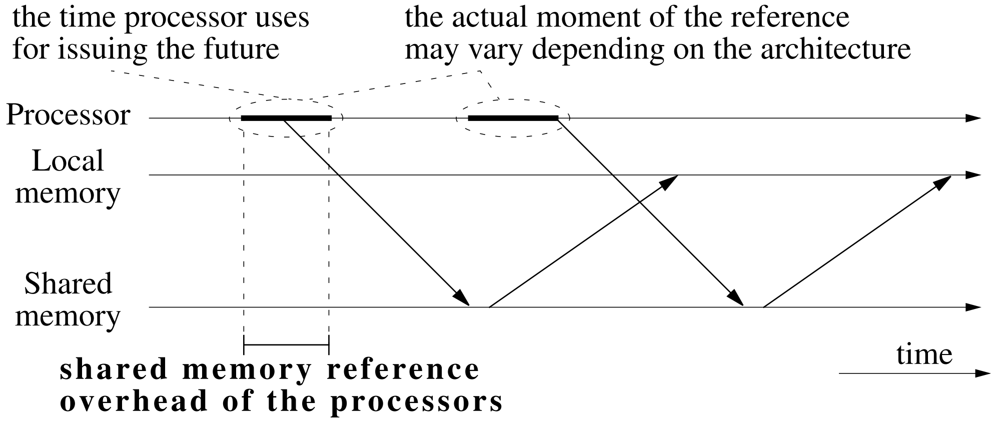


**Figure 4-6:** A diagram of the impact of overhead $B_P$.

cost of one future request (or write request) might get distributed to several processors, we shall assume that all processors communicate simultaneously.

**Definition 4.4:** The parameter $B_P$, the *shared memory reference overhead of the processors*, shortly overhead, stands for the time spent by a processor for accomplishing one shared memory reference in a situation where all (or most) processors are issuing references simultaneously (or nearly simultaneously).

Since it is likely that the initiating processor encounters most of the costs, we can use this measure also in the case when not all processors issue references simultaneously. The overhead is bounded by bandwidth inefficiency,

$$B_P \leq B \tag{4-2}$$

because if $B_P > B$, we could not communicate at frequency $B$. Note, that $B$ is principally induced by the network properties, and $B_P$ is principally induced by the processing node properties. Furthermore, the overhead is bounded by the latency,

$$B_P \leq L \tag{4-3}$$

because if $B_P > L$, the processor time for issuing the reference would be longer than the duration of communication, which is impossible in a situation where all processors communicate.

In reality, the costs of handling the messages distribute over a longer time, especially if the processors of the intermediate nodes have to participate in the communication. Modeling the distribution of the cost would be, however, too difficult. Consequently, we shall assume that the cost, i.e., delay, occurs entirely by the issue of the shared memory reference. This reasonable assumption does not affect the accuracy of the model since the model is asynchronous and the moment when a cost is charged makes no difference. Figure 4-6 presents the timing diagram of two possible future issues. The difference is due the work the processor has to do after the actual future request has left from the processor. In practice, however, the impact of the two situations is negligible, unless $B_P \approx L$.

Because of the relation of Formula (4-2) and because of the same unit, the overhead is used similarly as the basic bandwidth inefficiency $B$ parameter. If an algorithm does not interleave the communication and the computation, we only need to use parameter $B$ to compute the time needed for one communication. The difference occurs if the processors issue the shared memory references interleaved with the computations. In such a case the overhead slows down the computation independently of the bandwidth inefficiency. Hence, for each communication we need to add $B_P$ to the computation time. In practice, we do not have to issue as many concurrent shared memory references as in case $B_P = 1$. We shall discuss these algorithm design and analysis issues in more depth in Chapter 7.

### Block reference bandwidth inefficiency $B_B$

Using plain futures described above, the shared memory references transfer only one word. In some parallel computing systems, especially in circuit-switched systems, the starting time of each communication is considerable compared to the actual communication and its latency. Also, the initialization of the communication may reserve the whole bus. Further, in some packet-based systems, e.g., in the asynchronous transfer mode (ATM) interconnections, the packet size is fixed. Hence, the basic future packets would not result in full packets. Even the required one word header of the one word packet of the abstract F-PRAM future request results in only 50 % payload. Consequently, we should perhaps provide some possibilities to exploit the available bandwidth with better payload efficiency. A natural approach to improve the efficiency of shared memory requests is to allow block transfers between the shared memory and the private memories.

**Definition 4.5:** A *block future* of length *k* issues a future request to retrieve *k* consecutive words from shared memory to the local memory. Analogically, a *block write request* copies *k* consecutive words from private memory to shared memory.

The issuing of a block reference of length *k* takes

$$B_P \text{ or } k \times B_P \tag{4-4}$$

time steps on the issuing processor depending on whether the interconnection network interface operates autonomously and is connected to the private memory. The cost of Formula (4-4) is similar to the cost of the overhead $B_P$, which we defined to be the average cost over all referencing processors. In other words, even if the issuing takes only unit time, the later handling of the packets may take some time for the processor.

**Definition 4.6:** The parameter $B_B$, the *block reference bandwidth inefficiency* stands for the reciprocal of the average speed in words/cycle with which each long message proceeds through the interconnection network.

More accurately, the last elements of the vector of the future block of length *k* are expected to be written to the local memory in time

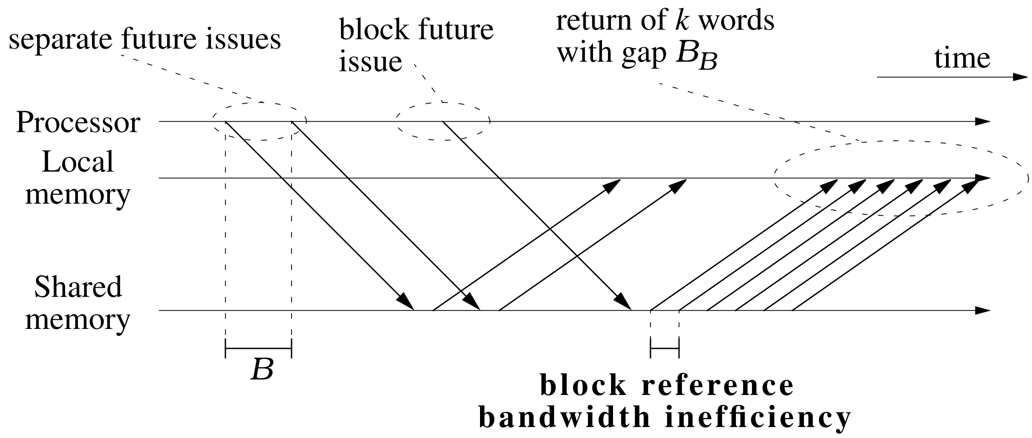


**Figure 4-7:** A diagram of the block future request and the parameter block reference bandwidth inefficiency $B_B$.

$$O(L + k \times B_B). \tag{4-5}$$

Figure 4-7 presents an example of two standard futures and a block future of length 6. The parameter $B_B$ impacts only in one direction for each request. The maximum frequency at which the processing nodes can issue block requests is rather difficult to define since the initialization time may or may not depend on the length of the message. To keep things reasonably clear, we shall state that the processors can issue ordinary and block shared memory requests with intervals

$$B + k \times B_B \tag{4-6}$$

without saturating the communication network. To keep the rule consistent with the earlier definitions, we shall state that $k = 0$ in cases of ordinary requests. The additional $1 \times B_B$ in a block reference of length 1 compared to an ordinary reference can be justified with the required additional word containing the length of the block reference.

### Single variable bandwidth $B_V$

A basic assumption of the F-PRAM model is that a single variable can be referenced once on every clock cycle. If the used parallel computer has a network that is able to combine the concurrent references to the same location [41], or multicast-capable memory modules, the single variable could be referenced by several, or all, processing nodes simultaneously without any additional delay, or with only a small additional delay. The definition of concurrent single variable accesses resembles the definition of the synchronization of sets of processors. We can easily model, and possibly implement, an all processor broadcast and a multicast to a single set of processors, but multiple distinct and concurrent multicasts to random sets of processors is more difficult to model. As opposed to the definition of the synchronization sets, we can define the referencing sets easily. The processors that make a reference to a given variable form a referencing set. The value of the variable is then multicast to the processors of the set. It is more difficult to define the cost of several sets of processors referencing simultaneously to several shared variables. The definition of this cost model would require inclusion of some knowledge of the structure of the interconnection network. That would make memory references too difficult to analyse in a general purpose and network-independent model. Consequently, we shall assume by default that a shared memory location can be referenced by only one processor on a clock cycle. We shall, however, retain a possibility for modeling machines which have more efficient shared memory by defining a secondary parameter for the purpose.

**Definition 4.7:** The parameter $B_V$, the *single variable bandwidth*, stands for the maximum number of references a shared memory location is able to serve in each clock cycle.

By default, the value of the parameter $B_V$ is one. If the shared memory of a machine can serve more references, the single variable bandwidth can be a larger constant, or a function of $P$, and in a very good machine it can be equal to $P$. In a not-so-good machine the parameter can also be less than one.[35]

In addition to the maximum number of concurrent references, we have to state whether there can be several different access sets active concurrently, or not. In other words, is the multiple-accesses-to-a-single-variable facility implemented via a single broadcasting network, or not. By default we assume that the number of reference sets is not restricted. The multiple accesses may only consist of read references. A write and a read, or several concurrent writes to the same shared variable yield an undetermined result.

### Synchronization delay $S$

The synchronization of the F-PRAM model is defined as a synchronization of the all processors in the machine. In other words, all processors are expected to execute the synchronization operation. After all processors have executed the operation, and after the knowledge of the synchronization has been propagated to all processors, the processors continue their executions.

**Definition 4.8:** The parameter $S$, the *synchronization delay*, stands for the time needed to propagate the knowledge of the synchronization completion to all processors, i.e., the time from the moment when the last processor executes the synchronization operation to the moment when all processors have begun to continue their executions.

Since the processors continue their executions independently, we do not require that the processors continue simultaneously. In other words, some processors may proceed very soon after the last processor has executed the synchronization operation, some others only after the $S$ delay. Figure 4-8 visualizes an example of the synchronization process.

The semantics of the synchronization emphasize that the synchronization is performed to ensure correct communication using shared memory references. Since the synchronization requires all shared memory references to be completed during the synchronization, the synchronization parameter $S$ cannot be smaller than the latency parameter $L$.

35. Actually, this is perhaps true for practically all current parallel machines.

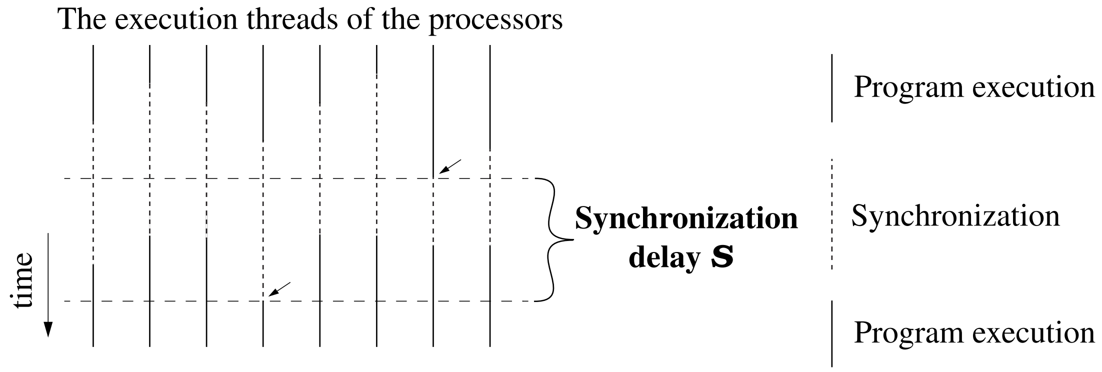


**Figure 4-8:** Synchronization delay of a set of 8 processors.

If a parallel computer has a dedicated synchronization network, we can assume that the synchronization can be accomplished quickly as long as we can ensure that the references get completed. In other words, in those cases we can assume that $S = \Theta(L)$, or even that $S \approx L$. If the parallel computer does not have any synchronization facilities, we can use Algorithm 7-9 to accomplish the synchronization in time $O(L \log P / \log L)$. The algorithm can also be used for submachine synchronization. If a parallel machine has a dedicated synchronization network, or executes synchronously, and it ensures the correct order of the shared memory references, we could allow $S < L$, but it would require an alternative definition of $S$.

A typical use of the synchronization delay parameter is to ensure that we do not use too much time doing synchronizations in a machine that does not have a dedicated synchronization network. It is not, however, as important and frequently used parameter as the latency parameter $L$ in the algorithm design phase. In most cases we can either assume $S = \Theta(L)$, or use $\max(L, S)$ in place of both of the parameters. The bigger role of the synchronization delay parameter is during the phase when we are estimating the execution time of an algorithm on a given machine.

### Single memory module bandwidth $B_M$, the number of memory modules $M$, and the data distribution scheme

The basic memory model of the F-PRAM model is the single monolithic shared memory model. The single multiport memory is, however, hard to build, and all current parallel computers use modularized shared memory or distributed memory. Therefore, we must remember that in most cases the shared memory will eventually be modularized. The use and the analysis of the modular memory structure is, however, so much more difficult, that we shall include it only as an option and with secondary parameters. By default we assume that the parameters $L$ and $B$ describe the memory module bandwidths accurately enough.

In a modularized memory system, all memory locations of the *virtual shared memory* are distributed among the memory modules. In a straightforward distribution scheme each location is mapped to exactly one module and all modules include an approximately equal number of memory locations. More complex schemes use, e.g., redundant copies of the memory locations in different modules, but we shall not discuss those methods here.

**Table 4-2:** Some possible combinations of parameters $B$ and $M$, and the expected effects to the probability of the module congestion.

| | $M = P$ | $M = C \times P$ | $M = P\log P$ |
|---|---|---|---|
| $B = 1$ | congestion | less congestion | no congestion[b] |
| $B = C$[a] | less congestion | little congestion | no congestion[b] |
| $B = \log P$ | no congestion | no congestion | waste of resources |

[a] $C$ stands for small constant, e.g., $M = 2$ or 4, if $B_M > 1$, then, e.g., $M = 2 \times B_M$
[b] unless poor hashing

Along the memory locations, also all the shared memory references get distributed among the memory modules. The main problem of the modularity is that several independent shared memory references fall on the same memory module simultaneously. We shall call such a situation, and such a module, a *hot spot*. An ordinary memory module is capable of serving only one reference on each clock cycle. As with the capabilities of a single location, we shall retain the possibility for modeling better memory modules by defining a pair of secondary parameters.

**Definition 4.9:** The parameter $B_M$, the *single memory module bandwidth*, stands for the maximum number of references a shared memory module is able to serve in each clock cycle.

As with the single variable bandwidth, the parameter $B_M$ is one by default. Depending on the machine it can be also a constant larger or smaller than one, or a function of $P$. The measure $B_M$ is not useful unless we also know the number of shared memory modules. Consequently, we shall define another secondary parameter to model the structure of the shared memory.

**Definition 4.10:** The parameter $M$, the *number of shared memory modules*, stands for the number of distinct memory modules, each of which can serve the $B_M$ shared memory references on each clock cycle.

If we consider the modularity of the shared memory at all, we shall assume by default that $M = \Theta(P)$. The parameters $B$ and $M$ have a special relation in the sense that we can adjust the probability of hot spots in memory modules by choosing them properly. Increasing either one will reduce the probability of congestion, but increasing $B$ reduces also performance, whereas increasing $M$ raises costs. Table 4-2 presents some combinations of the parameters and intuitive estimations on the probability of congestion. The estimations assume constant issuing of references and random distribution of the references. We shall leave a more accurate analysis of the optimal relations of the parameters $B$, $B_M$, and $M$ to future research.

### Data distribution guidance

The use and analysis of the modularized shared memory is hard unless we also have some knowledge of the mapping scheme of the shared memory locations to the memory modules. The mapping can be done either automatically by the runtime system in the parallel computer, or guided by the programmer. An automatic mapping can be done either with some deterministic mapping, e.g., in cycles or blocks, or with randomized mapping. In either case, the occurrence of hot spots is dependent on the usage pattern of the shared memory. A deterministic mapping may result in some good cases without hot spots, but will probably result in some extremely bad cases, in some applications, e.g., all processors might reference to the same module simultaneously. We can analyse a deterministic mapping only if we know both the mapping and the exact algorithm. The randomized mappings do not result in the optimal mappings, but they rarely result in the worst cases. If the randomization is good enough, we can use the expected values to achieve an analysis, which is reliable with a high probability. The drawback of the randomized (and cyclic) mappings is that they prohibit the use of the block references.

If we want to avoid the hot spots reliably and optimally, we have to define the data mapping ourselves within the algorithm. When choosing the mapping of a data structure of an algorithm, we have to know the usage pattern of the data structure within the algorithm. For optimal performance we have to choose such a mapping that avoids the hot spots. The programming model has to include directives that allow the programmer to guide the mapping of each data structure. In most cases it suffices if each dimension of each array can be specified to be distributed either cyclically, in blocks, or cyclically in blocks. The High Performance Fortran (HPF) standard proposal presents an example of the array distribution directives [47]. In the HPF, all vectors can be *aligned* according to, i.e., distributed according to, another array or a *template*, or an empty array. The arrays and templates are then aligned according to the *abstract processors*, which are finally aligned according to the physical processors by the compiler and the runtime system. We shall define distribution directives for the programming model of the F-PRAM model in Subsection 5.3.2.

### I/O bandwidth inefficiency $B_{IO}$

Some applications, such as digital video processing, require great amounts of data to be loaded and saved during the computation. Also some batch jobs might require a big input, or produce a big output. To ensure that the data of our tasks can be transferred to and from the parallel computer fast enough, we define a secondary parameter to model the I/O-bandwidth.

**Definition 4.11:** The parameter $B_{IO}$, the *input/output bandwidth inefficiency*, stands for the reciprocal of the I/O speed available for each processor. In other words, it tells us the frequency on which the processors can input or output data. By default, $B_{IO} = 1$.

Since the I/O bandwidth available for each processor at every clock cycle is typically very small, we defined the measure as the reciprocal, i.e., as inefficiency. Furthermore, the definition is more consistent with the other parameters. The value of the measure is likely to be a rather large constant, or a smaller constant multiplied by the number of processors in case of central I/O. The use of the parameter is to check how much the I/O time affects the execution of the whole program.

### 4.3.3 A sketch of the machine building cost

As we are considering a parallel computation model of real parallel computers, we have to remember that only a limited amount of resources are available. Besides the fact that our machine is not infinite, the limited resources usually show up in a form of less than perfect properties of the parallel machine. We discussed in Subsections 4.3.1 and 4.3.2 the methods of modeling such imperfections. In this subsection we shall discuss the reasons why such imperfections occur, and what we can gain by allowing such imperfections. The first-level gain is naturally money, since building a perfect[36] parallel computer would be prohibitively expensive, if not impossible. Building a less than perfect parallel computer can be much cheaper, and possibly the cheaper computer would be nearly as powerful as the perfect one of comparable size. Consequently, the secondary gain is that we can possibly build a larger but less perfect machine with the same money, and possibly get more computing power out of it. Our main subject in this subsection is developing some estimation techniques to be able to answer which features of parallel machines are worth their building costs in different situations.

As we have stated previously, estimating the cost of building a parallel computer in dollars is impossible, unless we bind accurately the type of the computer we are building. Moreover, the history of parallel computers has shown us that the estimation is difficult even with accurate plans [7]. Since so many different technologies are used in parallel computers, we cannot make even rough estimations on the costs of given components within our parallel computation model. Furthermore, we cannot reliably determine the relative costs of different parts of a parallel computer even if we knew their features. We can, however, make rough estimations of the costs or savings of improving or weakening a part of a parallel computer. Most importantly, we have to remember that there are no free components in a parallel computer. Even if the estimations are hard to make, we shall sketch here some candidates for the machine cost functions.

**Processing nodes**

Concerning the processing nodes of an F-PRAM, the cost of the processors is the easiest cost to estimate, $P$ processors cost approximately $C_{Pr}\times P$ dollars, where $C_{Pr}$ is the cost of one processor.[37] The private memory of the processing nodes has also a rather fixed cost $M_P\times C_{MW}\times P$, where $M_P$ is the number of words of private memory per processing node, and $C_{MW}$ is the cost of a word of memory. The cost of the input/output facility of the processing nodes depends on their capacity. We shall only state for completeness that the cost of the I/O facilities per processing node is $C_{I/O}$. Consequently, the cost of the processing nodes can be presented with formula

36. The definition of perfect naturally varies depending on the needs. Here we consider a computer perfect if it does not have any bottlenecks other than the processors, i.e., $L = B = S = B_{IO} = B_P = 1$, and $B_V = P$.

37. More accurately, the cost can be sublinear because of the relative reduction in design and factory founding costs. The difference is, however, usually irrelevant unless we design a custom processor for a unique parallel computer.

$$C_P = P\times(C_{Pr} + M_P\times C_{MW} + C_{I/O}). \tag{4-7}$$

The cost of the interconnection network interface it is not included in the formula because shall count it to be a part of the cost of the network.

### Interconnection network

The cost of the interconnection network is harder to estimate than the cost of the processing nodes because of the variety of different usable technologies. Here we shall ignore, e.g., optical interconnection networks and settle for fixed electric networks. An electric network consists of nodes and interconnections consisting of one or more parallel wires. The cost of the interconnections is probably nearly insignificant compared to the cost of the nodes. The cost depending of the length of the interconnections is probably unimportant;[38] and we shall assume that the relative costs of the interconnections depend mostly on the width of the interconnections. Moreover, the width of the interconnections affects strongly the cost of the nodes of the network, and the length affects the speed and the speed-dependent costs of the nodes. Consequently, we shall consolidate the cost of the network to the cost of the nodes. Depending on the chosen topology, the interconnection network may include routing-only intermediate nodes and shared memory module nodes in addition to the processor nodes. The multistage networks with routing-only inner nodes usually have the processing nodes and the possible separate memory modules in the outer nodes of the network. Hence, the processing nodes and memory modules might, for example, have lower degree than the inner nodes. Because the differences are, however, hard to estimate, we shall count the number of network nodes as the sum of the processing nodes, memory modules and the inner nodes, and charge the same cost for the all types of nodes. The clear cost parameters of the interconnection network are the number of nodes and the width and the degree of the nodes. The more difficult cost parameters are the speed, i.e., latency and the queuing facilities of the nodes. A possible cost function of the interconnection network would be

$$C_N = N_N \times C_{NW} \times C_{ND} \times C_{NL} \times C_{NQ}, \tag{4-8}$$

where

| | |
|---|---|
| $N_N$ | is the number of interconnection network nodes, |
| $C_{NW}$ | is the width cost of the nodes, |
| $C_{ND}$ | is the degree cost of the nodes, |
| $C_{NL}$ | is the latency (speed) cost of the nodes, and |
| $C_{NQ}$ | is the queue costs of the nodes. |

The costs are multiplied together because every property affects the others. The cost of the width of the nodes is a multiplicative factor since every component of the network must have the same width. Every addition in the degree of the nodes also requires its own set of components. The cost of the speed of the node is a multiplicative factor since every component of the network must be able to keep up with the others. The cost of the queues

38. The length affects, however, the maximum possible frequency on the wire.

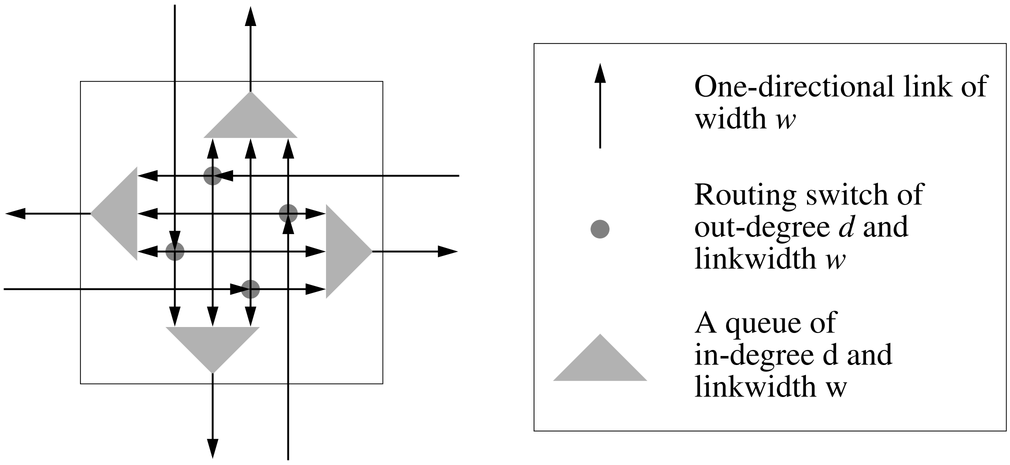


**Figure 4-9:** An example of a routing node of an interconnection network.

is multiplicative since the queues of the required width and length must exist in every direction of the interconnection network. Figure 4-9 presents an example of a routing node of, e.g., a 2-dimensional mesh. We can see that if we increase the degree of the node or the width of the links, the node will be more complex.

The linearity of the costs of the queues is valid for the real queues implemented directly, but less so for the queues implemented using memory and some logic. Therefore, we need to apply the next Formula (4-9) with different factor $AC_{NQ}$ depending which queue implementation the nodes use. Similarly for each cost, the doubling of a property will not probably double the costs of the property. Hence, each of the network costs is of the form

$$C_X = C_{X_0} + X_N \times AC_X, \tag{4-9}$$

where

| | |
|---|---|
| $C_{X_0}$ | is the base cost of the property $X$, |
| $X_N$ | is the quantity of the property in the node, and |
| $AC_X$ | is the cost of one additional unit of the property. |

The cost of the synchronization network of the F-PRAM is probably insignificant compared to the other costs. We can assume that the cost is dependent on the speed of the network and the number of processors. For completeness, we shall define the cost of the synchronization network as $C_S$. The cost is zero if the machine does not have a dedicated synchronization network.

### Shared memory

The cost of the shared memory depends heavily on the method it is implemented with. If the shared memory is distributed to the private memories of the processing nodes, it does not necessarily induce other costs than the increased size of the required private memory modules, and the increased demand of the bandwidth to the node. We must note, however, that using private memories for implementing the shared memory is probably less efficient than using dedicated shared memory modules. If the dedicated shared memory is implemented by modules and $B_M = B_V = 1$, we shall model the cost of the shared memory with formula

$$C_M = M \times (C_{MM} + N_M \times C_{MW}), \tag{4-10}$$

where

| | |
|---|---|
| $M$ | is the number of memory modules, |
| $C_{MM}$ | is the basic cost of a memory module, |
| $N_M$ | is the number of words in a memory module, and |
| $C_{MW}$ | is the cost of a word of memory. |

With the above basic assumptions, the cost of the shared memory is not directly dependent on the number of processors. Indirectly, however, the more processors we have, the more memory bandwidth and memory volume we need. Furthermore, the requirements for the properties of the shared memory are highly algorithm dependent, and the costs of an insufficient shared memory bandwidth occur within algorithm execution costs.

If $B_M > 1$ or $B_V > 1$ the costs are more complicated. The higher bandwidth to a memory module has to be implemented either by increasing the speed of the memory, which is usually unrealistic, by further dividing the modules to submodules, or by increasing the number of ports to the memory cells. The subdivision can be handled as an increase in the number of memory modules. The increase in the number of the ports to the memory cells increases also the parameter $B_V$. As estimated by Forsell [32], the additional cost of an implementation of the multiport memories is proportional to the square of the number of ports. Therefore, Formula (4-10) changes to the form

$$C_M = M \times (C_{MM} + N_M \times C_{MW} \times B_V^2), \tag{4-11}$$

which implicitly allows also $B_M = B_V$. If the single variable bandwidth is improved by using a combining interconnection network,[39] the cost of the shared memory is unchanged, but the cost of the interconnection network grows. We shall not try to estimate those costs in the form of a formula. Instead, we notice that the nodes of a combining interconnection network need to have fast comparison facilities and enough memory to be able to store the information of the combined messages. Furthermore, the combining network does not increase the single memory module bandwidth unless all references are destined to the same location.

### Total machine cost and algorithm execution cost

We defined in Section 4.2 that the components of the F-PRAM work independently unless they have a reason to interact. Consequently, the building cost of an F-PRAM is the sum of the costs of its components, i.e.,

---

39. We shall present the combining interconnection networks briefly in Subsection 4.6.2.

$$C_{build} = C_P + C_N + C_M + C_S. \tag{4-12}$$

For completeness we also take into account the maintenance costs of a machine during its lifetime. Hence, the lifetime costs are

$$C = C_{build} + C_{maint} \tag{4-13}$$

which, however, does not provide us any new viewpoints to parallel computing. We defined the total cost because we need to be able to state the cost of a parallel computer to be able to properly examine the execution costs of the algorithms. Using the algorithm cost model with experimental constants we can approximate the time used by an execution. Even without the constants we can compare the running times of two or more algorithms. By combining the algorithm costs and the machine costs, we get the cost of the execution of the algorithm in dollars.

**Definition 4.12:** The *ultimate execution cost* of a parallel algorithm $A$ is

$$C_A = \frac{T_A}{T_C} \times C, \tag{4-14}$$

where

| | |
|---|---|
| $T_A$ | is the execution time of the algorithm $A$, |
| $C$ | is the lifetime cost of the parallel computer, and |
| $T_C$ | is the lifetime of the parallel computer. |

Using Formula (4-14), we could try to minimize the ultimate cost of executing an algorithm. Naturally, we are not able to state the exact cost of the execution in dollars, unless we had extremely accurate information on the cost of the components of parallel computers and a very detailed plan of the parallel computer. Likewise, we seldom can estimate the lifetime of a parallel computer. Furthermore, the formula does not take the execution time requirements into account. Hence, the resulting optimal machine might be the cheapest home computer sold at the time. We could count the execution time also and using the above definitions, we could define a cost optimal architecture for a given algorithm by finding the optimal cost-performance combination of all properties and the time requirement of the task. Because the optimal combination would, however, probably be too difficult to find, we shall not even try to suggest it. Moreover, there is little value in finding a cost optimal architecture for a single task.

Instead of trying to use the formulas directly, we can check the properties one by one, by inspecting which changes of the properties were worth their cost. We have to settle for comparing the proportional costs of changes. The analysis of F-PRAM algorithms often yield complexities which depend directly on one parameter, for example, bandwidth inefficiency. These bottleneck properties are the ones we should first check for possible cheap improvements. For example, if halving the bandwidth inefficiency $B$[40] would speed up the computation nearly twice, but cost only 10 % more, the improvement would be

40. That is, doubling the bandwidth.

very profitable. Besides the improvements of the machine, we can check the algorithms also against weakening of the machine. For example, if the multiport shared memory costs one third of the cost of a parallel computer, but we could write nearly as efficient algorithms without it, we could give up the multiport memory, possibly in favor of, e.g., more overall bandwidth, or more processors.

To formalize the use of a single parameter at a time, we shall define a desirable feature of parallel algorithms, namely the scalability with respect to the properties.

**Definition 4.13:** If the ultimate execution cost (4-14) of an algorithm does not increase when we change a parameter of a parallel machine, we shall state that the algorithm is *cost scalable* with respect to that parameter.

For example, when doubling the number of processors, and other dependent properties, the execution time should drop to half since the cost of the computer doubles. This concept of scalability with respect to a single property at a time is probably the best we can do without deeper knowledge of the costs of building parallel computers. We shall later briefly discuss the use of scalability, but mostly we shall leave it to future research.

As a summary of this subsection, we can state that the route of finding a cost optimal architecture is probably difficult and that the results are valid for one task only. Therefore, we should find a architecture that is a reasonable compromise between the cost and the desired properties. The requirements for the properties can be found, e.g., using the F-PRAM analysis and/or the F-PRAM emulator system presented in Chapter 6. On the other hand, the F-PRAM analysis helps us to find suitable algorithms for the current existing parallel computers.

## 4.4 Rationale of the choice of the parameters and the structure of the model

In this section we shall give the reasons for the F-PRAM model to exist. To begin with, we shall conclude the facts that led to the chosen level of accuracy of the definition of the model. Then we shall describe the features in more detail. As we noted earlier, we chose the shared memory model for both programming easiness and portability. The message passing model on a fixed noncomplete graph network would not have been portable and easily programmable in the general case. The complete network message passing model is more portable, but nearly as hard to implement as shared memory. One can argue on the comparative programming easiness of shared memory and message passing models, but we feel that the message passing model is the more process-oriented model and the shared memory model is the more data-oriented model. Since our problems usually involve mostly processing of a lot of data, we chose the data parallel shared memory approach.

We defined the F-PRAM model rather accurately with a rather large set of parameters. The main reason for the accuracy is not to restrict the computation, but to provide a versatile model for different types of parallel computers. The number of the parameters prevents using any loopholes while designing algorithms. Consequently, the algorithms should really be efficiently executable. Furthermore, the secondary parameters allow us to exploit the possible extra features of some parallel computers.

We gave the reasons for most of the features of the F-PRAM model along with their definition, but in this section we shall make some more refined notes on them. In Subsection 4.4.1 we shall compare the F-PRAM model with other similar models, and tell the reasons why a new model is required. The rest of this section includes more technical rationale of the model. Especially we shall discuss how parallel computers can implement the F-PRAM, or, vice versa, how we model parallel computers with the F-PRAM.[41] We shall consider the implementation of an F-PRAM machine, the implementation of the processing nodes, and the implementation of the futures in Subsections 4.4.2, 4.4.3, and 4.4.4, respectively.

### 4.4.1 Comparison with existing parameterized models

We presented some of the existing parameterized parallel computation models in Section 3.3. More exact simulation comparisons between the existing models and the F-PRAM model will be given in Subsection 4.7.2. In this subsection we shall emphasize the reasons for the differences and the features which make the F-PRAM model preferable compared to the earlier models.

The most obvious feature of parameterized parallelism models is naturally the set of parameters on each of the models. The problem of finding an optimal number of parameters has been recognized by several researchers. The bigger and more accurate the set of parameters is, the more difficult is the analysis of algorithms if we take all of the parameters into account. The favorite parameters of the previous models have been the number of processors, the communication latency, the processor communication overhead, and the network bandwidth. The F-PRAM model and the LogP model [28] include these four parameters in one form or another. Most of the other models include a subset of these parameters. The F-PRAM model also includes several other parameters. Later in this chapter, Table 4-3 in Section 4.7 presents a quick comparison between the F-PRAM model and a few other parameterized models. As the set of parameters is large, the analysis of F-PRAM algorithms may be rather difficult. While this is not entirely false, the division of the parameters in two groups, and the guidelines to use one parameter at a time definitely help the use of the model, as we shall see in Chapter 7.

The F-PRAM is a rather concrete model compared to the other models. Especially the programming model for the F-PRAM, which we shall present in the next chapter, is a complete programming language. Consequently, the F-PRAM models both the machines and the programming. An advantage of the accuracy of the model is that we have been able to implement an emulator of the model. The advantage of the emulator is that we were able to gather measured information on the real executability of the algorithms. Furthermore, the values of the parameters can be more accurately determined as we have more detailed information on their semantics of the model. For example, the fact that the packets are two words long and are delivered between the processors and the shared memory tell us more than the usual statement "the processes communicate." A disadvantage of the concreteness of the model is that it might bind more features than necessary. Especially this applies to the concrete programming model to be defined in Chapter 5, but we

41. We must remember that we defined the F-PRAM model to model existing and forthcoming parallel computers, not to be an example of parallel computers.

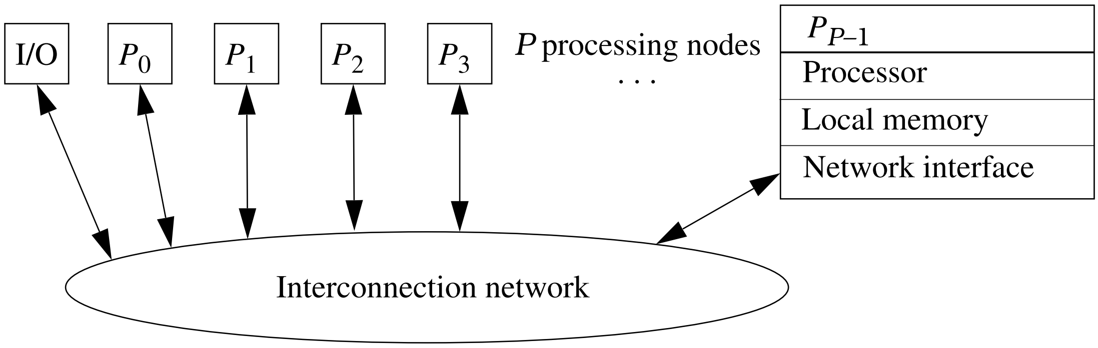


**Figure 4-10:** A possible implementation of an F-PRAM. The shared memory is implemented within the local memories. I/O is centralized.

must remember that the F-PRAM model does not imply any particular programming model. The rather large set of parameters of the F-PRAM model still allows fair versatility within the model, and especially on the parallel machines which can be modeled using the model. We must retain to use only those parameters that are needed in the analysis.

### 4.4.2 The structure of the F-PRAM model

We defined the conceptual structure of the F-PRAM model and the F-PRAM machine in Section 4.2, more accurately in Figure 4-1 (page 36). The conceptual structure does not restrict the implementation of the F-PRAM model in any way. In other words, we can model different types of parallel computers with the F-PRAM model, which is naturally a requirement for a general purpose model. Of the conceptual components of the F-PRAM model, the dedicated synchronization network is the easiest one to omit. The barrier synchronization can be performed using either a library routine, or Algorithm 7-9 (page 160). Furthermore, the shared memory can be, or more often, will be, implemented by distributing the memory elements among the processing nodes. Either the processing nodes have a separate memory bank dedicated to the shared memory, or the shared memory slice is included in the private memories of the processing nodes. Figure 4-10 presents the structure of the F-PRAM without the synchronization network, shared memory, and parallel I/O. The machine of the figure resembles the structure of most of the current parallel computers, but we feel that the model of Figure 4-1 is better since it can be used also to model computers with dedicated synchronization network and shared memory.

### 4.4.3 The structure of the processing nodes

The block diagram of Figure 4-1 only tells us the list of the components of the processing nodes of the F-PRAM model. Each of the components has its own function within the node and each of the components needs to cooperate with one or more other components within or outside the node. Figure 4-11 shows a coarse block diagram of the components and connections of a possible implementation of an F-PRAM node. The optional (gray arrow) network interface–private memory connection is needed if the shared memory is implemented by distributing it into local memories of the processing nodes or if we want

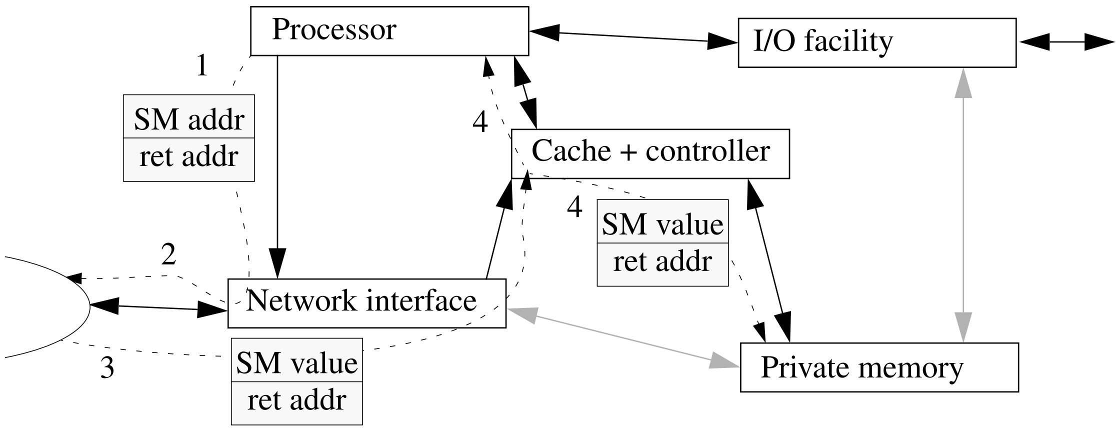


**Figure 4-11:** A possible implementation of an F-PRAM node, and the route (1-4) and the content of a future request.

to implement block futures efficiently. The I/O facility–private memory connection is needed for possible direct memory access (DMA) input/output, but it is irrelevant with respect to the F-PRAM model. As the processing node probably has a cache for the private memory, the network interface can return the results of the resolved futures directly to the cache since it is very probable that the processor will need the result rather soon. We do not need a local cache for the values of the shared memory since all shared memory reads will be performed using futures.[42] The actual value is a read from the result of the future, which resides in the private memory.

If we implement the shared memory by distributing it among the local memories, then the load of the local memories increases with some amount. For each future request, there will be four local memory references: a write of the place-holder to the local memory, a read at the remote "shared" memory location from where the value is read, a write at the local memory where to the value of the future is written, and finally the actual read made by the processor. The amount of shared memory requests can be estimated to be a fraction of all memory requests, or to be a fraction of all instructions. The fraction is highly application dependent and input size dependent, but is likely to be rather small in most cases, e.g., either in the range 1-10% of all instructions, or in the range 1-50% of all memory references.

### 4.4.4 Implementation of futures

A packet used to resolve a future needs to include the local address of the future, i.e., the private memory location where the value is to be returned. Additionally, while the packet is on its way to the memory module containing the requested value, the packet needs to contain the address of the requested memory location. While the packet is returning to the requesting processor, the address is not needed anymore, but the requested value naturally is. Thus, a future-resolving packet consists of two words, each of which must be able to

42. Nevertheless, the local caching of the shared memory would be rather easy since the F-PRAM model does not guarantee the consistency of the results of the shared memory references unless the machine is synchronized.

contain the address of any memory location in the system, the other must also be capable to include a word of data. Similarly, a write-request packet to the shared memory requires two words, the destination address and the value. Consequently, all packets that are being delivered are two words long. Additionally, a bit or two might be required to distinguish the different types of packets from each other. It should be, however, possible to encode these bits to the address field of the packet unless the word length is equal to the logarithm of the address space. If the addresses take the whole word, we would need a third word consisting of the bits and possibly the number of the destination processor.

Keeping the local work of the processors minimal should not be difficult since the complexity of the future issue should not be much harder than issuing a local memory request. The only difference is the inclusion of the local memory address to the request. Currently, a processor of the Cray T3E can issue a shared memory write in 19 clock cycles and continue its execution long before the actual write occurs hundreds of clock cycles later [26]. By reducing the synchrony of the communication between the processor and the support circuitry, the initiation of the short messages could probably be even faster. Even if there is no apparent reason for it, the current hardware of the T3E does not, however, support the asynchronous shared memory reads. The reason of this design choice has probably been to reduce the risk of inconsistency between the local memory, caches, and the stream buffers.

## 4.5 Efficient algorithm design and analysis methods for the F-PRAM model

Even sequential programming is considered sometimes difficult. Parallel programming even for the plain PRAM model is more difficult. Hence, one could argue that the F-PRAM model is too difficult to be programmed well. In this section we shall discuss the methods which clarify the programming and algorithm design process. In other words, we shall discuss not only efficient use of the processors, but also efficient use of design time.

While designing algorithms, we cannot usually analyse them on the fly, but still we should keep the forthcoming analysis in our minds. This will be useful especially when designing optimal and portable algorithms. In practice, this means first analysing the problem, and then programming using those F-PRAM parameters that are the most important for the problem. While analysing the problem we should not think in terms of any particular solving method since that would possibly give us a wrong idea of the complexity of the problem. This distinction of the fundamental properties of the problem and the properties of algorithms is, however, difficult. The most essential task in the analysis of the problem is the analysis of data movements during the process of resolution of the problem. Besides the data movements, we naturally have to analyse the operations to be done. Compared to more traditional parallel algorithm design, in the F-PRAM algorithm design, the data movement analysis is more important since the model charges higher cost for the shared memory references. Naturally, when designing optimal algorithms, we have to take both computation and communication into account, and, based on the analysis, decide which one is the more important one to minimize on each stage of the algorithm. Furthermore, when designing portable algorithms, we have to guess the possible combinations of machine properties. Then we can build in the algorithm some logic to minimize the most important costs on run time.

### From problem analysis to algorithm analysis

The analysis of a problem has two phases. Firstly, as in sequential algorithm design, we have to clarify the operations required to deterministically achieve the result from the inputs. Secondly, we have to check which of the operations can be executed in parallel, and plan the best method of parallelizing them. If we have enough processors available, the execution time of the algorithm is the number of consecutive stages in the parallelized code. If we do not have enough processors available at some stage, the existing processors need also to work for the missing processors, which usually prolongs the execution. Depending on our goal and the resources available, we try to minimize the execution time using a given number of processors, the execution time using as many processors as needed, to maximize the efficiency of the algorithm, or usually to fill both the first and the third goal. These goals are the same as the goals when designing algorithms for the unified-cost parallel computation models, e.g., the PRAM model. We presented a classification of the speedups of the algorithms in Chapter 3. The difference is that the nonconstant costs of some operations make the counting of the execution time more difficult. For the algorithm designer this means that there are more choices and optimization possibilities to be taken into consideration. If some operations are very expensive, we may have to circumvent their use by using cheaper operations. If the parallelization of a relatively small problem or subproblem is very expensive, we should possibly perform the task sequentially. Especially, when considering the efficiency of the execution of the relatively small tasks we should consider the sequential option. On the other hand, we may have a fixed number of processors fully assigned to the task, for which cases we only should optimize for execution time.

### Optimizing data movement

As we stated earlier, the analysis of the data movements is rather important when designing algorithms for the F-PRAM model. Also, most of the F-PRAM parameters are related with the costs of the data movement in relation to the cost of atomic local operations. Concerning data movements, our goals are usually to minimize the data movements, distribute the data movements evenly enough to avoid congestions, and make the data movements concurrent with the computation. These goals concern especially the bandwidth inefficiency parameter $B$, which dictates the minimum interval between the shared memory references. If the time spent for the shared memory references is short even with the extra waits caused by the bandwidth inefficiency, we usually can accept it. If the time is long, or if we want optimal solutions, we have to try to find some useful local operations to be done between the shared memory references. Similar situation occurs with the latency parameter $L$, i.e., we should find something useful to do for the processors while the future is being resolved. The latency is, however, easier to hide since, in addition to local operations, the processor can also issue other shared memory references while waiting for a future to be resolved. We must remember, however, that the parameters $L$ and $B$ both affect simultaneously, i.e., we cannot issue several shared memory references to hide the latency unless we have enough bandwidth available to do so. Since in some bus-based computers the latency is smaller than the bandwidth inefficiency, we cannot use the several concurrent shared memory references issued by a single processor for latency hiding in those cases.

Concerning the secondary parameters of the F-PRAM model, perhaps the shared memory reference overhead $B_P$ is the most important one. If the overhead $B_P$ is greater than one, the processors have less time for local operations between the shared memory references. The single variable bandwidth $B_V$ is at most one in practically all existing computers.[43] Therefore, we probably should not use it when designing portable algorithms. The parameters concerning modularity of the shared memory and blocked shared memory references are important for final optimization of algorithms for particular computers, but we shall leave the analysis of their use to future research. The same applies to the parameter I/O bandwidth $B_{IO}$.

In practice, when designing algorithms for most problems, we can adapt to a large range of values of latency $L$ and the number of processors $P$ without losing any of the optimality. In other words, they can be hidden rather easily. On the other hand, the bandwidth requirements are often inherent in the problems, and the possible restrictions in bandwidth inefficiency $B$ cannot be overcome easily. Thus, when designing an algorithm, we usually first consider only parameters $P$ and $L$, and only afterwards check the parameter $B$ by determining a requirement for it for the algorithm to be able to execute optimally. Additionally we shall determine its impact on the running time of the algorithm in a case when it does not fulfill the requirement. If the impact is too large, we should perhaps try to find an alternative algorithm which would possibly use less bandwidth, or distribute the bandwidth usage more evenly over time. If this is not possible, we should consider either allowing the algorithm to be less efficient, or try other possibilities such as using less processors, which often helps for the bandwidth shortage. Similarly, we should check the rest of the relevant parameters for possible conflicts against machine properties, and possibly modify our algorithm correspondingly.

### Asynchrony

A big difficulty in designing algorithms for the F-PRAM model is the asynchrony of the processors and especially the shared memory references. For every communication, i.e., for every write-read pair of a single shared memory location, there should be some synchronization to ensure that the order of the references will be correct. The easy solution is to divide the computation into stages within which no location is both written and read,[44] and to include an explicit barrier synchronization between every stage. The drawback of this approach is that if the synchronization cost $S$ is large and the synchronization is needed often, the efficiency of the algorithm may decrease significantly. In those cases we have to rethink the synchronization needs and synchronize only the communications between the processors. In practice this can be done either using counters and/or locks

43. The known exception is the IBM RP3 research computer, which was modeled after the NYU Ultracomputer project. The RP3 uses combining of messages for achieving a greater bandwidth to a single variable [7].

44. Or written several times.

associated with the data, or using special not-yet-available values on the variables or processes. The use of counters is a rather convenient method of ensuring that the processors are in the same stage, but we have to be carefully ensure that all the processors that use the counter update it correctly. In practice it is easy to maintain a dedicated counter for each processor and allow other processors to read it. It is more difficult to set up several processors to update the same counter or lock.

In some cases the difficulties induced by the asynchrony of the F-PRAM model can be turned into advantage. Since the asynchrony induces undeterminism, the F-PRAM model suits well for undeterministic randomized algorithms. We have to remember, however, that the fact that the order of the shared memory references is not guaranteed does not induce true randomization but usually machine dependent and algorithm dependent patterns of the order of the references.

### Analysis of the F-PRAM algorithms

The basic analysis of F-PRAM algorithms begins similarly as the analysis of sequential algorithms, by counting the operations. The difference is that the operation count must be done for the worst case over the processors between each synchronization. Furthermore, synchronizations are charged the cost of $S$ operations, and the shared memory references the cost of $B_P$ operations, which, however, is one by default. After the traditional operation counting, we also have to check the program for any additional delays caused by the shared memory references with respect to the rest of the F-PRAM parameters. Considering the latency $L$, the time used between a future issue and the use of the value is $\max(X, L)$, where $X$ is the time needed for the operations between the two points. If either $B$, $B_V$, or $B_M$ form a restriction for the communication, they appear to increase the latency of the shared memory references. The estimation of the apparent increase of a random latency is, however, difficult. Therefore, we have to analyse the number of references per processor and shared variable within a given time, and check whether the machine is able to serve the references or not. If the total bandwidth required by all references, or references to a single variable, is too large, the running time has to be reanalysed according to these requirements. The running time of a program block which issues and uses shared memory references is thus

$$\max\left(T, L + \max\left(\frac{Data \times B}{P}, \frac{Data_V}{B_V}, \frac{Data_M}{B_M}\right)\right), \tag{4-15}$$

where

| | |
|---|---|
| $T$ | is the basic running time of the block, |
| $Data$ | is the total number of shared memory references, |
| $Data_V$ | is the maximum number of references to a single variable, and |
| $Data_M$ | is the maximum number of references to a single memory module. |

The above analysis results in either a complex function of most of the parameters, or a simpler function of fewer parameters and additional requirements on the rest of the parameters. The function of all parameters is probably prohibitively complex for most algorithms. Consequently, we should choose to the cost function only those parameters which affect the performance most, and attach the rest of the parameters as restrictions to the cost function. In practice, all the parameters which induce only a rather loose requirement of the property should not be included in the main cost function. Furthermore, another good criterion for the choice is the resulting apparent complexity of the main cost function.

As the above discussion on running times showed, the traditional analysis method of adding costs of operations and multiplying with the number of iterations takes into account only a part of the F-PRAM parameters. This method naturally forms the basis of an analysis of an F-PRAM algorithm, but it is not sufficient. After the basic analysis, we have to check the possible delays caused by the possible references to the results of the futures before the results are available. We have to check the sufficiencies of the whole machine, single variable, and I/O bandwidth within each phase of our algorithm, and possibly reevaluate the complexities on those phases, until we get a stable running time with no violations of any of the F-PRAM restrictions.

A “change” in the complexity of a phase of an algorithm might lead to a change in the tuning of the algorithm. For example, if we notice that a body of an outer iteration takes more time due to the bandwidth requirement than we originally thought, we might be able to reduce the number of iterations in the inner iteration that forms the body of the outer iteration. We should use this feedback from the analysis to improve the flexibility of the algorithm by inserting into it some logic to take into account the performance-related machine property. This will complicate the algorithm, but should not be overly difficult when done stepwise. Besides, we have already analysed the new feature.

## 4.6 Options and optional restrictions within the model

A good part of the interesting options for a model of parallel computation are included in the secondary parameters of the F-PRAM model. Since we reasoned about the secondary parameters earlier, we shall not discuss them here again. Instead, we shall discuss some currently rejected features that might be interesting or useful in some applications. We shall define the new features only informally, and discuss the reasons why they should/should not be included in the model. The common problem of several of the following options is that they would require all processors to execute the operation synchronously.

### 4.6.1 Dynamic change of the parameters

We defined the parameters of the F-PRAM model either as machine-dependent constants, or as machine-dependent functions of $P$, and required that the values of the parameters are constants during the execution. An interesting generalization of the parameters would be to allow them to change during the execution of a program. If the parameters can change during the execution, we can write algorithms that optimize themselves to the changes of the machine during the execution. The algorithms will, as we shall define in Chapter 5, be able to use the parameters to adapt to the machine. If the parameters would change, the algorithms would change their adaptation during the execution.

Naturally, the physical properties of a parallel machine change rarely unless the machine encounters a fault. Even if the fault tolerance is an important issue in some parallel applications, we shall mostly ignore it in this thesis. More frequent change than a physical fault is the change in the load of the parallel computer. Because parallel computers are rarely dedicated to execute only one task at time, the load induced by the other programs may affect the execution of a single program. The accurate modeling of these changes caused by other tasks is practically impossible. Consequently, we are bound to rely on our existing set of parameters that model the features of a parallel computer. For this reason, a change-over in the load of the computer causes the values of the parameters to change as we were changing the whole computer to another one with slightly different properties and parameters. If an algorithm is able to adapt to changes, it may choose to change its behavior if the change in the machine is significant.

Since the parameters are used during the execution only by the algorithm to optimize its own behavior, e.g., the lengths of iterations, the change of the parameters need not to be continuous or immediate. Furthermore, all processors must have exactly the same values of the parameters all the time. Otherwise the distributed use of the parameters would be impossible. Since the changes are not needed often, and since they need to be done synchronously, we could specify that the parameters get updated during the synchronization of the whole machine. If an algorithm uses a value of a parameter beyond a synchronization point, it needs to save the original value in an ordinary variable. Alternatively we could require that the parameters will not change unless separately requested to be updated. The request could be done by calling a library routine that evaluates the new values for the parameters using some gathered performance data.

### 4.6.2 Combining network

Broadcasting a single value to all processors is an important operation in many algorithms. In the F-PRAM model the operation is realized by reading the value from the shared memory by every processor. Reading a single variable is restricted with parameter $B_V$, single variable bandwidth, which by default is one. Thus, the broadcasting has to be performed using a tree-like algorithm to avoid the serialization of the references. The serialization of $P$ references is a severe delay, unless $B_V = O(P/L)$. As we shall see in Section 7.5, broadcasting will take time

$$O\left(\frac{L \log P}{\log L + \log B_V}\right), \tag{4-16}$$

which is asymptotically optimal, but has a rather large constant compared to the easiness of the broadcast problem.

If the nodes of an interconnection network would have some logic, they could detect the simultaneous[45] future-requests of the same memory location, save the information of one on them, and forward only the other one. Consequently, the memory module would get only one request, which would be easy to serve. On the way back, when the packet

45. The requirement of simultaneity of the references makes the combining less useful in the asynchronous F-PRAM model.

gets to a node where the original requests were combined, the packet gets duplicated to two messages according to the information saved by the node. The combining-capable interconnection facility is possible, although expensive, to build [7]. A straight advantage of the combining system is that it is practically the only possible solution to achieve $B_V = P$. Furthermore, the combining reduces the total traffic in the interconnection network, and especially hot spots around the memory module that includes the value to be broadcast.

The message-combining system can operate independently of the processors, especially it can work independently of the possible asynchrony of the processors. Since it makes the synchrony-requiring tree-like broadcasting algorithm unnecessary, it can considerably reduce the number of synchronizations required in an asynchronous F-PRAM algorithm. We shall not, however, include the option separately in the F-PRAM model, since setting $B_V = P$ is a close enough approximation of the combining network.

### 4.6.3 Vector operations

Many scientific computations consist of mostly vector operations. These operations can usually be implemented in different parallel computers rather efficiently. Because the implementations are highly machine-dependent, they are not always easily portable. Furthermore, the optimizations might be too difficult or even impossible to be done by an application programmer with a high-level language. The solution has been to define subroutine libraries which can be used by the application programmers and which can be implemented efficiently by the computer manufacturers. A good existing example of such a library is the BLAS library for matrix and vector operations [68]. A general purpose parallel programming model should not include as large set as the BLAS does, but instead only the most elementary operations.

The more primitive vector operations can be divided to three groups: reduction, distribution, and scan/prefix [16, 65] operations. The reduction operations return a property, e.g., logical OR, sum, product, etc., of a vector. The distribution operation is the duplication of a value to every element of a vector. The scan/prefix operations perform a sequence or *read-op-write* operations for a vector. All of these have been suggested as primitives of parallel computations models. Moreover, the shared memory libraries of many parallel computers include optimized versions of these operations [26]. Also, the MPI library of message passing computation includes a set of these operations between the processors a communication group.

The problem in the use of the vector operations in the F-PRAM model is that their execution would require synchronization of the processors. The synchrony requirement is natural, but it is inconsistent with the asynchronity of the F-PRAM model. The advantage of the vector operations would be that even added with the synchronization cost, they could be performed more efficiently using dedicated hardware than using the F-PRAM model primitives. Especially the constants of the complexities would be probably considerably lower. Furthermore, if a parallel machine has some dedicated hardware for vector operations, we can assume that it also has dedicated hardware for synchronization, which reduces the possible synchronization cost.

### 4.6.4 Read-modify-write operation

The latency and asynchrony of the shared memory references make secure updating of a shared variable rather difficult if it is used by several processors. Even if a processor is the only one that updates the variable, the processor has to read the old value using a future, update it locally, and issue a write request to the shared memory. Things get much more difficult if there are several processors that try to update the same variable, and all of the updates must survive. Asynchronous readings, updates and writings by several processors would cancel the effect of a part of the interleaved updates. The solution is to lock the variable for the use of only one processor at a time. This would, however, need a rather complicated and time-consuming protocol. The protocol includes attempting to lock, checking the success of the attempt, read, update, verifying the completion of the update, and releasing the lock. The whole process takes at least $6L$ time for each update, which is too much if every processor wants to update the variable every now and then.

If we have a well defined set of processors that is performing the read-modify-write operation simultaneously, we could also use a parallel algorithm to perform the resulting prefix-operation in parallel. For example, if the processors are performing additive updates, we can perform them in parallel with a prefix-sum operation. The problem is, as we noticed in the previous subsection, that the parallel prefix-operations require rather strict synchrony.

Since the performing of read-modify-write using the standard F-PRAM is hard, we could add a primitive read-modify-write request to the F-PRAM model. The operation of the read-modify-write request would be similar to a future request, except that the read-modify-write request would have the updating function with it. The atomic read-modify-write operation is slightly inconsistent with F-PRAM model because it conceptually requires logic in the shared memory. If, however, we implement the shared memory by distributing it into the private memories of the processing nodes, we could use the local processors to actually perform the read-modify-write. Thus, we would not need any additional hardware for the operation. The network interface should be able to interrupt the processor to do the update, though. This implementation will naturally affect the performance of the processors, more accurately it should show up in the parameter $B_P$. Besides the use of the data synchronization and prefix operations, the read-modify-write operation is not so vital within the basic algorithms that we would include it in the F-PRAM model.

#### Replace-future

If we cannot include the read-modify-write operation in the model because of the requirement of logic in the shared memory, we could alternatively exploit an almost equally useful operation, the *replace* operation. The replace operation would be a combination of future and write-request operations. A *replace-future* would include the value to be written to the shared memory, and return the original value to the processing node that issued the replace-future. The replace-future would be very useful in situations where exactly one of several processors may receive a single value from a single memory location. As opposed to the read-modify-write, the replace-future would not require the shared memory to be able to perform any calculations.

The Cray T3E SHMEM library supports both an atomic swap and an atomic increment operations on a remote memory. As the get implementation of the SHMEM library, also these operations are of a blocking nature, i.e., the processors have to wait until the operation completes. There is not a clear reason for this as the processors are idling meanwhile.

### 4.6.5 Guaranteed shared memory reference latencies

We defined the semantics of the shared memory references so that the latencies are only the expected worst latencies, not the worst possible latencies. Hence, the model does not guarantee the completion order of different references made by even the same processor. The completion of the future requests is naturally easy to detect, but the completion of the write requests cannot be detected otherwise but by polling the location with future requests until the written value is found from there. The problem occurs when another processor writes a new value to the location meanwhile, and the original writer cannot differ whether the write-request is still in transit or not.

We considered guaranteeing a write and a read to the same location by the same processor with gap $L$ to take their effects in the original order. In other words, if a processor writes a value, waits $L$ time units and issues a future to the location, it would not result in the old value of the memory location. Finally, however, we never could have benefited from this fact when writing the sample programs, thus, we left the guarantee out. Moreover, the guarantee might be difficult to maintain if the interconnection network saturates, and several requests are queued to the intermediate nodes. This would be especially obvious if we used randomized routing.

## 4.7 Matching the F-PRAM model with the existing models

In this section we shall relate the new F-PRAM model to other existing parallel computation models. We shall relate models by comparing the cost models and by presenting some simulation techniques and algorithms between the models. Since the F-PRAM is a model of parallel computers, and also a model of parallel computations, the simulation of the F-PRAM by another model is a slightly misleading concept. If the F-PRAM is able to model a computer of another model, the simulation is not a simulation in a strict sense. Referring to the F-PRAM is not accurate enough, we have to state also the parameters before we can examine the ability of an instance of the F-PRAM model to simulate another model of parallel computing. Consequently, we shall discuss in this section how the F-PRAM is able to model other parallel computer models, and what types of F-PRAMs are able to simulate different parallel computation models.

### 4.7.1 The PRAM model and the F-PRAM model

The PRAM model is the most researched theoretical shared memory model of parallel computing. Since the F-PRAM model and the PRAM model have rather similar structures, as we can see by comparing Figures 3-1 and 4-1, the comparisons between the models are rather straightforward. We can characterize the PRAM trivially with F-PRAM parameters $L = 1$, $B = 1$, $B_P = 1$. The parameter $P$, the number of processors, is not as clear since PRAM is usually defined to have as many processors as needed. Obviously a $P$-processor PRAM is a $P$-processor F-PRAM. Furthermore, a $P$-processor F-PRAM can simulate a PRAM having more than $P$ processors, as we shall see in Subsection 5.2.1. Also, the synchronization cost $S$ is not clear. By usual definitions, the processors of a PRAM execute instructions synchronously, but can branch to different instructions of the same program. To some extent, the processors can be kept in synchrony by inserting dummy instructions into the less time-consuming branches of the program [60]. This approach, however, does have its limitations, and in some cases we have to use the explicit barrier synchronization, which will take $O(\log P)$ time using Algorithm 7-9. Consequently, we shall state that $S = O(\log P)$, even if we might be able to avoid a part of the synchronizations. The parameter $B_V$, the single variable bandwidth, depends on which PRAM flavor we are discussing. Because the EREW-PRAM allows only one concurrent read to each memory location, $B_V = 1$. The CREW-PRAM does not restrict concurrent reads to memory locations, thus $B_V = P$ in case of CREW.

The simulation of a PRAM requires the ability of referencing the shared memory on every clock cycle, and synchronization after the execution of the local instructions. Consequently, the simulation of one PRAM step takes $O(L+S)$ F-PRAM steps, which is slow and inefficient, unless $L = S = 1$. To be able to perform work-optimal simulation, we have to simulate a more-than-$P$ -processor PRAM with a $P$-processor F-PRAM.[46] Using the previous time needed for one step, we have to simulate a $P\times(L+S)$-processor PRAM with the $P$-processor F-PRAM. Therefore, each of the F-PRAM processors have to perform the operations of $L+S$ PRAM processors, which takes $O(L+S)$ time. Furthermore, the shared memory references get completed within time $O(L)$. Finally, all processors have to synchronize, which takes time $S$. In total, the simulation step takes $O(L+S)$ time, which is both time, and work optimal. As we remember from Subsection 4.3.2, usually $S \geq L$, thus $O(L+S) = O(S)$. On the previous analysis we did not take into account the bandwidth restrictions of the F-PRAM model. The PRAM simulation uses up to $O(1)$ bandwidth/processor/clock-step, since every operation can be a shared memory reference. Therefore, we require an F-PRAM having $B = O(1)$. In practice, the ratio of local operations and shared memory references is either a constant, or a function of the size of the input, which often is a function of the number of processors. Consequently, we can have algorithm-dependent exceptions to the previous requirement on the bandwidth inefficiency. Finally, we shall notice that simulating a CREW-PRAM requires $B_V = O(P)$, but for simulating an EREW-PRAM it suffices $B_V = O(1)$. No F-PRAM can simulate efficiently a CRCW-PRAM using any straightforward algorithm, since we explicitly forbid the simultaneous writes to a shared memory location.

### 4.7.2 Matching the F-PRAM model with other parameterized models of parallel computing

As we stated earlier, the simulation of machine modeling techniques is a slightly contradictory concept. Hence, in this subsection we shall rather compare the parameter sets and

46. The use of *parallel slackness* is an essential part of all latency hiding simulations [66, 104].

**Table 4-3:** A comparison of the F-PRAM model and a few other parameterized parallel computation models. Unified parameter notation similar to [39].

| model | communic. method[b] | synchrony[c] | number of processors | bandwidth restriction[d] | latency measure | overhead measure | block transfers | other parameters |
|---|---|---|---|---|---|---|---|---|
| F-PRAM | SM | AB | $P$ | $B$ | $L$ | $B_P$ | $B_B$ | $B_V, B_M, M, S$ |
| BSP [104] | MP | BS | $P$ | $g$ | $L$ | | | |
| LogP [28] | MP | AM | $P$ | $g$ | $L$ | $o$ | | |
| LogGP [6] | MP | AM | $P$ | $g$ | $L$ | $o$ | $G$ | |
| QSM [39] | SM | BS | $P$ | $g$ | | | | |
| Y-PRAM [99] | SM | LS | $P$ | $B$ | $L$ | | | |
| HPRAM [46] | SM | AB | $P$ | | $L$ | | | $S$ |
| LPRAM [3] | SM | LS | $P$ | | $L$ | | | |
| Phase LPRAM [38] | SM | BS | $P$ | | $L$ | | | $S$ |
| Interval model[a] [71] | SM | BS | $P$ | $I$ | $I$ | $I$ | | |
| BDM [58] | SM | BS | $P$ | | | | $Bsz$[e] | |
| PRAM [34] | SM | LS | $P$ | | | | | |

[a] A set of parameters given by Maggs, Matheson, and Tarjan.
[b] SM = shared memory, MP = message passing, DM = distributed memory.
[c] LS = lock-step/common clock, BS = bulk-synchrony, AB = asynchronous, barrier synchronization at request, AM = asynchronous, synchronization via message passing.
[d] $B$ = bandwidth inefficiency (soft limit on issue rate, eventual effect on completion of the communication), $g$ = gap (hard limit on issuing rate)
[e] $Bsz$ = block size of the communications

other features of the models. We introduced these models earlier in Section 3.3. Within each of the models, we shall consider matching in both directions. Firstly, we shall "simulate" the other model with the F-PRAM, i.e., characterize the other model, and its parameters, in F-PRAM parameters. Secondly, we shall present the opposite, i.e., model the F-PRAM with the other model. Because the parameter sets are of different sizes, we have to ignore some of the parameters. Consequently, we shall refer to these modelings as *projections* between the models. Some of the following models use originally message passing between the processors rather than shared memory. With respect to simulations between the models, this subject is rather irrelevant. We shall discuss in Subsection 4.7.3 the simulation between the message passing models and the shared memory F-PRAM model. In this subsection we only consider the parameters and the cost functions of the models. Since the names of the parameters of the different models appear similar, we shall include subscript tags in the text for some of the other than the F-PRAM parameters to avoid possible mix-ups. Table 4-3 presents a quick comparison between the F-PRAM model and a few other parameterized models.

**BSP/XPRAM model**

We shall first express the XPRAM model [103] in F-PRAM parameters. The XPRAM model is a realization of the more general BSP model. The XPRAM defines the number of processors like the F-PRAM model does, thus $P = p_{\text{XPRAM}}$. The latency $L$ of shared memory references equals the global operation time $L_{\text{XPRAM}}$ of the XPRAM model, thus $L = L_{\text{XPRAM}}$. The synchronization cost of the XPRAM model equals the latency cost, thus $S = L_{\text{XPRAM}}$. The time of global operations $g$ in the XPRAM corresponds to the overhead parameter $B_P$ of the F-PRAM model. As the XPRAM model does not take bandwidth of the message router into account, we need to assume that $B = B_P$. The rest of the features defined by the secondary parameters of the F-PRAM are ignored by the XPRAM model. Therefore, we have to assume the default values of the F-PRAM parameters.

A $P$-processor F-PRAM can also simulate a $p$-processor XPRAM. Since the parameter $L_{\text{XPRAM}}$ of the XPRAM model includes time for both latency and synchronization, we have to choose $L_{\text{XPRAM}} = L + S$. The global operation cost corresponds with the processor communication overhead, thus $g = \max(B, B_P)$ The rest of the F-PRAM parameters are ignored in the XPRAM model.

**LogP and LogGP models**

The LogP model [28] resembles the F-PRAM model more closely than any of the other previously mentioned models. The conceptual difference is the message passing approach of the LogP model. Also, the LogP model does not have a separate synchronization primitive. The similarity is that every parameter of the LogP model is also included into the F-PRAM model. Consequently, the latency parameters $L_{\text{LogP}}$ and $L$, the overhead of communication $o_{\text{LogP}}$ and the processor communication overhead $B_P$, the gap $g_{\text{LogP}}$ and the bandwidth inefficiency $B$, and the number of processing nodes $P_{\text{LogP}}$ and $P$ correspond with each other, respectively. Since the LogP model does not have a dedicated synchronization mechanism, we have to use the synchronization Algorithm 7-9, which we shall present in Section 7.5. Hence,

$$S = \frac{L_{\text{LogP}} \log P_{\text{LogP}}}{\log L_{\text{LogP}}}, \tag{4-17}$$

unless $g_{\text{LogP}} \geq L_{\text{LogP}}$, in which case $S = g_{\text{LogP}}$.

The LogGP model [6] adds to the gap per byte (G) parameter to the LogP model. The parameter G corresponds to the $B_B$ parameter of the F-PRAM. The slight difference is that LogGP does not account for transferring the length of the message. Therefore, LogGP defines the length of a send to be $(w \times k - 1) \times G + o$ cycles instead of the $k \times B_B + B_P$ cycles of the F-PRAM model. The $w$ stands for the width of a word in bytes.

***Y-PRAM* model**

The *Y-PRAM* model [99] allows for the recursive division of the machine in independent submachines. Since the F-PRAM model does not require division, a whole $P_{\text{Y-PRAM}} = 2^p$-processor *Y-PRAM* equals a $P = P_{\text{Y-PRAM}}$-processor F-PRAM. The latency

$\delta(S_{\text{Y-PRAM}})$ and bandwidth inefficiency $\beta(S_{\text{Y-PRAM}})$ parameters of the *Y-PRAM* model correspond to the respective parameters of the F-PRAM model. Using the whole machine of size $P_{\text{Y-PRAM}}$, we get $L = \delta(P_{\text{Y-PRAM}})$, and $B = \beta(P_{\text{Y-PRAM}})$. The synchronization cost of the *Y-PRAM* model is based on the minimum length of the phases, thus $S = \delta(P_{\text{Y-PRAM}})$. The secondary parameters of the F-PRAM model have to be assumed to be their defaults since the *Y-PRAM* model does not include them.

When simulating the *Y-PRAM* model with the F-PRAM model, the F-PRAM cannot enforce the independency of the submachines, but checking can be done by the compiler. Since the *Y-PRAM* computation proceeds in phases, after which the submachine is synchronized, we have to include the explicit submachine synchronization in each phase. Using Algorithm 7-9 we can synchronize the submachine of size $S_{\text{Y-PRAM}}$ in time $O(L \log S_{\text{Y-PRAM}} / \log L)$. The *Y-PRAM* parameters are functions of the size of the submachine, but they have to be projected to constants. Consequently, we get $\delta(S_{\text{Y-PRAM}}) = O(L \log S_{\text{Y-PRAM}} / \log L)$ and $\beta(S_{\text{Y-PRAM}}) = B$ for every $S_{\text{Y-PRAM}} \in [1..P]$.

### 4.7.3 Simulations of message passing models

Even if the shared memory and the message passing approaches of parallel computing appear very different they can simulate each other with reasonable effort. After all, unless a separate monolithic memory is used, the shared memory must be implemented distributed. Hence, the references to the shared memory are implemented using messages between the processors and the memory modules. In this subsection we shall informally sketch the methods how an F-PRAM can simulate a message passing system, and vice versa.

We shall present a protocol to simulate a $P$-processor message passing system with a $P$-processor F-PRAM by using $d \times P$ shared memory locations as mailboxes. The $d$ stands for the in-degree of the nodes of the message passing system. Thus, a completely connected system would require $P \times (P-1)$ memory locations. We shall consider synchronous communication, where a processor receives/sends from/to one named processor at a time only. For each connection from processor $a$ coming to a processor $b$, there exists a shared memory location $C_{a,b}$ which originally includes a special[47] “not-receiving” value *InAct*. When the processor $b$ wants to receive a value from the processor $a$, it first checks the completion of the possible previous communications by reading the memory location and waiting until it contains the value *InAct*. After that, it writes a special “ready to receive” value *Rec* to the memory location $C_{a,b}$, and starts polling until the location contains any other value than *InAct* or *Rec*, which will be the value to be received. After the processor $b$ has got the actual value, it writes yet another special value *Got* to the location, and continues its execution. The processor $a$, which wants to send the value $V$ to the processor $b$ first waits by polling the location $C_{a,b}$ until it contains value *Rec*, and after then writes the actual value $V$ to the location $C_{a,b}$. After writing the value, the processor $a$ waits to be sure that the processor $b$ has read the value by polling the location $C_{a,b}$ until it becomes *Got*, after which the processor $a$ writes the original value *InAct* to the location, after which the communication is complete. Algorithm 4-1 presents the previous algo-

47. If no special values can be reserved for these purposes, we need to use two memory locations, first of which includes the tags and the second one the actual values.

```
proc send(dest, val)
    t := 0;
    while(t <> Rec) do
        future t := C[dest, PID];
    fwrite C[dest, PID] := val;
    while(t <>Got) do
        future t := C[dest, PID];
    fwrite C[dest, PID] := InAct;

proc receive(src) : value
    t := 0;
    while(t <> InAct) do
        future t := C[PID, src];
    fwrite C[PID, src] := Rec;
    while(t = Rec or t = InAct) do
        future t := C[PID, src];
    fwrite C[PID, src] := Got
    return t;
```

**Algorithm 4-1:** Message passing using futures.

rithm using the algorithm notation to be given in Chapter 5. The communication takes $O(L+B)$ time after both processors have started it. A more straightforward algorithm using only one "received" tag would not work since the F-PRAM model does not guarantee the order of shared memory references. The problems would appear when the receiving processor would start to receive again before the sender has detected the earlier receive. The above algorithm works well also for the situation where each processor is communicating concurrently with several other processors. The different pollings of each directions just should be interleaved. In practice, we should be extremely careful of making sure that each of the concurrent communications would get completed before we leave the communication phase of the program. Asynchronous communication with queues would require a more complex structure with separate locking protocol for the queue-pointers. As a summary of our sketch of simulating the message passing model with the F-PRAM model we can state that the simulation would be much easier using the fair latencies or the read-modify-write primitive, which we discussed in Subsections 4.6.5 and 4.6.4, respectively.

The simulation of F-PRAM using a message passing distributed memory model resembles the simulation of any shared memory model by the message passing model. We shall first consider an asynchronous message passing system with logical all-to-all connections. Each of the processors of the message passing system simulates an F-PRAM processor and is responsible on maintaining a slice of the shared memory. Since the F-PRAM model allows asynchrony of the processors and memory references, the simulation need not proceed in global phases. Consequently, the efficiency of the simulation does not require overloading, i.e., none of the processors has to simulate more than one processor. When a processor makes a shared memory reference, it computes the address and the owner of the location using the hash function and the sends either the future request or the write request to the processor which owns the location. After the send, the processor proceeds its own computation. Among the execution of the local operations, the processor is interrupted for incoming messages to serve them before executing the next local operation. The processors receive three types of messages, write requests, future requests and returning future results. The write requests are obviously written to the local part of the shared memory. The results of the future requests are read from the local part of the shared memory and sent to the processors that made the requests. The returning future results are written to the private memory, and they are available when they are actually needed. The communication-capability parameters of the F-PRAM can be adjusted by setting the amount of communication each processor does between the local computations. If each of the processors serves at most $x$ request per $y$ clock cycles, we get $B_M = x/y$. The ratio is probably less than one, which means that the processor does not serve requests on every cycle. The bandwidth inefficiency and the latency parameters depend directly on the properties of the used communication network.

If the message passing system does not allow all-to-all (virtually) direct connections, the processors also have to perform the routing of the messages. We shall consider a network with diameter $l$, and assume that the decision on where to send a packet going to a given node can be done in unit time. When a processor has or receives a message that is destined to another processor, it forwards it unless it has already sent a message to that direction on the same cycle, in which case it queues it. Each of the packets gets handled by $O(l)$ nodes, consequently we get $L = O(l)$ and $B_P = O(l)$, assuming that the other F-PRAM parameters guarantee that the network does not saturate.

# Chapter 5

# A programming language for F-PRAM model

The serial random access machine (RAM) model of computation does not bind the programming model accurately. Very different programming models and languages, such as FORTRAN, Smalltalk, and Lisp, can be rather efficiently compiled to the RAM model. Particularly, we can usually estimate the costs of the programs rather easily and reliably, no matter which programming model we use. Furthermore, the different programming models are efficiently portable to any machine implementation of the RAM model even if the RAM implementations differ rather much. The case of parallel computation is not that easy, as there is an additional nontrivial scope to model.

The F-PRAM model is probably too complex to be used as a general purpose "bridging model"[48] between hardware and programming, but we still have similar goals. The F-PRAM model tries to model different parallel computers, but it is also intended to serve as a basis for different programming models. In other words, we can define several programming models that can be compiled to parallel machines that are modeled with the F-PRAM model. The F-PRAM model of parallel computation does not determine the used programming model accurately. To be able to present and implement algorithms we shall, however, present an example programming model. In addition to the plain algorithm notation, we define a language to be able to write and run full programs for the F-PRAM model. Additionally we shall give guidelines for analysing the algorithms. We shall call this new programming model *F-PRAM Programming Model* (FPM). Another interpretation of the acronym could be *Future Parallel Modula* as the language is a subset of Modula-2 with parallelism primitives and futures for shared memory access. In this chapter we shall define the programming model, the algorithm notation and the language concurrently. Our algorithm notation is only a compacted version of the full programming language. The programming model only defines the available parallel programming primitives.

A vital feature in any programming model or language is that the code should be easily readable. An extremely compact presentation of programs, as seen in some functional programming models, is not desirable if even the author of the program has difficulties examining his or her own programs. A program written with a good programming

48. As named by Valiant [104].

model should be readable even by a person with some knowledge of parallel programming, but little or no knowledge of the particular model. Since we are interested in performance of the programs, and since the performance cannot be estimated unless we are able to examine the functionality, i.e., the execution, of the programs, we shall use an imperative language presentation instead of a declarative one. For the same reasons we shall rule out the functional languages, even if they have some additional values being of higher level and safer.

As the F-PRAM model itself, also the FPM model has two levels of constructs. The basic programming tools that are required to write parallel algorithms correspond to the primary parameters of the F-PRAM model. The secondary parameters of the F-PRAM model can be used during the analysis phase to check the details of the executability of the algorithm. Some of the secondary parameters can be included in the programming model to make programs that adapt to different computers even better than using only primary parameters. This is useful if we can model some exotic features of some parallel computers with the secondary parameters, and if the use of a feature speeds up our algorithm considerably. On the other hand, because the use of the secondary parameter usually complicates the design of portable algorithms, we should first try to make the basic algorithms, and only after that possibly try to modify them to make use of the secondary features.

In this chapter we shall first describe the basic programming paradigm in Section 5.1 and the basic parallelism structures in Section 5.2. As with the base model, we shall give some alternatives for some parallelism structures of the basic programming model in Section 5.3. We shall give the examples of the actual use of the programming model in Chapter 7.

## 5.1 Basic paradigm

The skeleton of the new FPM programming model is the traditional Pascal/Modula -like algorithm notation that is used in many variations in most texts on serial and parallel algorithms. We shall use a rather low level variant of the notation, for example, we shall use explicit iterations with loop variables instead of referring to all members of a set. The notation is strictly imperative, and relies on a small set of program composition constructs. The constructs include variables of different types, arrays, assignments, serial composition, conditional structures, iteration and procedures. The FPM model adds only the parallelism structures to the basic notation. One significant change induced by the parallelism structures is that the FPM model does not guarantee deterministic execution of our programs. Otherwise the appearance, or the design and analysis methods, of parallel FPM algorithms should not differ too much from the appearance of traditional serial algorithms.

The execution of FPM model programs uses the so called "*Single Program, Multiple Data*" (SPMD) approach. In other words, every processor executes exactly the same program code. While executing it, they behave similarly and perform the same operations by default. The difference between the processors is the unique processor-id of each of the processors. Additionally, each of the processors has it own private memory, and possibly private I/O facilities. The difference of the use of these features, however, is induced by the use of the processor-ids.

The SPMD approach in the FPM programming model is slightly modified and implicitly expressed. The program code appears to be serial by default. Within the serial body of the program there are parallelism constructs that seemingly invoke the parallelism for the parallel parts of the program. However, when we execute the parallel program, even the serial parts of the program are executed by every processor. The processors have distinct local copies of all private variables, and the processors make the local computations using the local variables. The exception to this seemingly contradictory rule of executing the serial code in parallel is that the writes to the shared memory within the serial part are executed by one processor only. The parallel sections of the program only renumber the same processors. The renumbering mostly includes assigning each processor an iteration variable that is used instead of the processor-id within the parallelized loop to differ the actions of the processors. We shall discuss this parallelism structure in more detail in Subsection 5.2.1.

The basic FPM programming model gives the programmer a tool to use the F-PRAM model rather directly. Especially the programming model does not add restrictions on the use of the model. Thus, the programmer has the responsibility for the structure and the correctness of the program. The programming model itself does not enforce any methods for correct programming. Therefore, we recommend that a programmer either uses some safe approach to ensure the correct behavior of the program, or tries to reason accurately the correctness of the possible unsafe portions of the programs.

The main source of the undeterminism of the FPM programs is the asynchrony of the processors and especially the asynchrony of the shared memory based communication. The FPM programming model does not guarantee the order of a write and a read to and from the shared memory unless there is some synchronization between the references. Similar undeterminism problems naturally occur if we make several writes to the same shared memory location without ensuring the correct ordering. The synchronization to ensure the correct order of the references can be done either using a barrier synchronization of the whole machine, or by ensuring that the written value has arrived at its destination and then synchronizing the communicating processes pairwise. The latter more fine grained synchronization can be used when only a small number of processes are communicating with each other, and the rest may execute independently of the local communication.

The model does not prevent the programmer from making any questionable pairs of references. Only the programmer is responsible for including enough synchrony between the processors to ensure that the communication, and the whole algorithm, is correct. Unguaranteed memory references are not automatically disastrous for the correctness of the program, but the result of every questionable reference has to be checked carefully against errors. In some undeterministic randomized algorithms we can use the undeterminism of unsynchronized memory references to bring some more undeterminism to our algorithm. Still we have to remember that we may not assume any truly random phenomena from the system. For simplicity, we recommend that the processors, and the data they have written, have to be synchronized before any communication through the shared memory.

When using well designed parallel computers, we can usually make some assumptions on reasonable fairness of communication. For example, we could assume that a reference that was made $10L^2$ cycles ago has been completed without ensuring it by synchronizing the processors. The FPM programming model does not, however, give such guarantees, since it is definitely a bad programming habit to rely on such assumptions. In any case, a synchronization of two processors, or a data exchange between two processors, takes only $O(L)$ time by using time-tagged variables, or the Algorithm 4-1. Therefore, the asynchrony should not be too severe restriction.

## 5.2 Parallelism structures

In this section we shall define the parallelism structures that modify the standard algorithm notation to the algorithm notation for the F-PRAM model. The necessary structures include primitives for management of parallelism, for communication, and for synchronization. The parallelism management appears to create the parallelism to the algorithms, but in reality it is only processor management. Our communication medium is the shared memory, which we shall handle via shared variables. The basic synchronization structure is the barrier synchronization. Additionally we shall sketch the pairwise synchronization of processors and data. In addition to the basic parallelism structures, we shall define a set of read-only variables that provide the parameters of the F-PRAM model for the algorithm designer and for the algorithm. Here we shall give the parallelism structures of the FPM language.

### 5.2.1 Management of parallelism

The FPM model requires that all processors execute the whole body of the program. Hence, we would not necessarily need any special parallelism-creating constructs, other than a processor-id (*PID*) for each processor. The processor-id is, however, too awkward structure to be used for high-level programming. Since we usually use the processors to parallelize iterative loops, it is useful to renumber the processors according to the loop indices. In other words, to assign each processor a new number that corresponds to its task within the parallelized loop. We shall express this new number easily with a loop variable. The parallel loops will be initiated with a *par-do* statement that resembles the basic serial *for-do* -structure, but has some restrictions and special features. We must note here again that even if the *par-do* statement appears to invoke new processors, in reality it only assigns a new value for the iteration variable of each of the processors. These distinct values of the iteration variables then differ the behavior of the processors within the *par-do* statement. The syntax of the plain *par-do* statement is

**par** iter_var := low **to** high **do** (5-1)
  *statements*;
**end**;

which means that each of the processors gets assigned distinct values of the local variable *iter_var* within the range *low..high*, and each of the processors executes the body *statements* of the parallel loop asynchronously. The processors executing the statements with the same value of *iter_var* form a *thread*. Consequently, there will be $high-low+1$ threads. The boundary expressions *low* and *high* must be evaluable by every processor independently, and they must produce the same values for every processor. In practice, the boundary expressions must consist of only private copies of shared or common variables. The independency requirement allows the processors to execute *par-do* statement asynchronously. Some processors even might have completed the *par-do* statement before some other processors begin it.

Since all of the processors execute the same program, we have to transform the *par-do* statement to a serial structure to be executed by each processor. Each of the processors evaluates the iteration variable *iter_variable* using its own *PID* and the boundary expressions. The previous simple *par-do* statement can be executed as

**for** iter_var := low+*PID* **to** high **by** *P* **do** (5-2)
  *statements*;
**end**;

which automatically adapts the number of iterations to be executed by each of the processors to the total number of iterations to be executed. If the $P < (high-low+1)$, each of the processors execute $\lceil (high - low + 1)/P \rceil$ or $\lfloor (high - low + 1)/P \rfloor$ iterations of the *statements*. Consequently, one or more processors get assigned two or more different values of *iter_var*. The same obvious parallelism statement implementation was used in *pm2* language for the PRAM [60].

The implementation (5-2) does not work well if $P > (high-low+1)$ and we have nested *par-do* statements. A part of the processors would skip the whole statements and the increased opportunity parallelism of the inner *par-do* statements would be wasted. If the range $high-low+1$ of the outermost *par-do* statement is smaller than the number of processors *P*, we have to use several processors to execute each of the $high-low+1$ iterations, and especially the possible inner *par-do* statement. More accurately, we have

$$P_{new} = \max\left(1, \frac{P_{old}}{high - low + 1}\right) \tag{5-3}$$

processors to be used for each "iteration" of the new *par-do* statement. Thus, we can rewrite the implementation of the *par-do* statement in the form

**for** iter_var := low+*PID*/max(1, *P*/(high–low+1)) **to** high **by** *P* **do** (5-4)
  *statements*;
**end**;

which assigns the same values of *iter_var* to the threads of $P_{new}$ processors. Each processor of each thread gets assigned a local new processor-id $PID_{new}$, which ranges from 0 to $P_{new}-1$. The new *PID* is unique within the thread but not between the sets. The set of processors within a thread should behave like the processors executing the serial part of the whole program. In practice, each processor of the tread should perform all operations except the writes to the shared memory. One processor must, however, perform even the writes to the shared memory as in the serial part of the whole program.

We shall denote the depth of nested *par-do* statements we are executing with variable $D$, and the number of processors usable in level $D$ with function $P(D)$, and the $PID$ of a processor at level $D$ with function $PID(D)$. When a set of processors executing the body of a *par-do* statement encounters an inner *par-do* statement, it will behave exactly like the whole set of processors when it encountered the outermost *par-do* statement. The difference is that there are only $P(D)$ processors available, each having a local processor-id $PID(D)$ in the range $0..P(D)–1$. For generality we shall also define that $P = P(D)$ and $PID = PID(D)$, i.e., we refer to the available processors per thread. We shall give a more accurate definition of the practical implementation of the nested *par-do* statements in Section 6.2.

The basic version of the *par-do* statement (5-1) does not synchronize the processors in any way. If we want the processors to be in synchrony after the *par-do* statement, we have to add a synchronization statement after the *par-do* statement. If we want to synchronize the processors in the middle of a *par-do* statement, we either have to divide the *par-do* statement into two parts and insert the synchronization there, or be sure that no processor executes more than one iteration of the *par-do* statement, and just insert the synchronization into the middle of the *par-do* statement. Since the processors execute the whole body of the *par-do* statement before the possible next iteration, a synchronization in the middle of a body of the *par-do* statement which is iterated several times by each processor does not work.

**Local (private) variables**

The local variables of the FPM model can be more accurately called "private variables for the $P$ processors." Hence, the private variables are not private for the iterations of the *par-do* statements.[49] The same private variables of a processor are used at each iteration of the *par-do* statement the processor performs. Thus, when using a private variable across several *par-do* statements or across synchronization, we cannot assume that the value of the variable to be preserved unless we are using at most $P$ parallel iterations. If we want to preserve values across several *par-do* statements which have more than $P$ parallel iterations, we need to store the values in different elements of shared or private arrays. When using only *par-do* statements within range $0..P–1$, the private variables remain private to the iterations of the *par-do* statements.

### 5.2.2 Shared variables

The processors of the F-PRAM model communicate by writing and reading to and from the shared memory. Our FPM programming model expresses shared memory in the form of shared variables, and writing and reading to and from the shared memory by references to the shared variables. We introduce the shared variables similarly as normal private variables, but with the keyword *shared* in front of the definition. The processors will create their own copies of each of the private variables to their local memories, but the shared variables will be stored in the shared memory only. Since the shared memory is monolithic in the basic F-PRAM model, we do not initially have to know the method of storing

49. Which resemble parallel threads.

the variables in the shared memory. We shall describe the option of the modularized shared memory in Section 5.3.

The F-PRAM model states that the writes to the shared memory are asynchronous and the reads from the shared memory are realized as future issues and local reads of values of the futures. The FPM programming model adapts this convention unchanged. The writes to the shared variables appear similar to the writes of any variables, we just may not assume anything about the completion time of the write request. The only way to read shared variables is to issue a future, which loads the value of a shared variable to a local variable. The syntax of a shared memory reference using futures is the modified assignment statement

**future** local_variable := shared_variable; (5-5)

which assigns a special not-yet-available value[50] to *local_variable*, and issues the future that will resolve the value of *shared_variable*. Both the write issue and future issue take constant time by default. If we consider the processor communication overhead $B_P$, also the references of the FPM model take time $B_P$. After a shared memory reference was issued, we can expect that it will be completed in time $L$. The processor does not receive any notice of the completion of a shared memory write. The completion of a future causes the not-yet-available value to be replaced by the value that was read from the shared memory, but does not affect the execution of the processor. After the resolving of a future is completed, the local variable that was used as a placeholder, and that was assigned the value of the shared variable, appears to the processor exactly as any local variable. If a processor tries to use the local variable before the completion of the future, it stops until the value of the future arrives. This blocking use of futures is adequate for our purposes in most algorithms since we can use the value of $L$ to ensure long enough iterations before attempting to use the value.

### Nonblocking future observation

The future mechanism makes a reference to an unresolved future to halt until the future is resolved. In a practical implementation, the processor has to repeatedly check the future until it becomes completed. We do not want, however, to disturb the programmer with the danger of reading the unusable not-yet-available value. Instead, we shall require that the nonblocking reference to a future can be used only within a special boolean checking expression. The syntax of the checking expression is

**fcheck**(fut_variable) (5-6)

which returns the value *true* (i.e., future has been served) or *false*. In practice, the expression will be used as a part of a conditional *if* or *while* statement. For example, the statement

50. The use of a special not-yet-available value instead of a separate tag reduces the number of values that can be stored in a word by one, but allows us to use any variable as a placeholder for a future without using any additional words to hold the tag information.

**future** fut_variable := ... (5-7)
...
**while**(**not fcheck**(fut_variable) **and** *something_useful_to_do*)
*do_something_useful;*
*use* fut_variable;

waits for the future *fut_variable* by doing something less urgent meanwhile. In most algorithms, however, the *something_useful* may be as urgent as the *use* of the *fut_variable*, and, therefore, we could as well perform it fully before the use of the *fut_variable*.

**Handling of concurrent references**

Since the processors and the interconnection network of the F-PRAM model are asynchronous, we cannot predict which of the shared memory references occur concurrently at the shared memory modules. We cannot even predict which of the references were issued simultaneously. Consequently, we shall not forbid concurrent references, but we shall state that they will be serialized unless $B_V > 1$. The serialization of concurrent reads, i.e., future requests, is easy to define and produces predictable results whether the order of the references is preserved, or not. The serialization of the concurrent writes is also easy to define by stating that the order of a concurrent write and another reference to the same variable is random, unless they are separately synchronized. In other words, the effects of the successive shared memory references may occur in any sequence, unless we ensure the order via synchronization.

**Block transfer operations**

We defined the blocked shared memory references as an extension to the standard future in Subsection 4.3.2. The incorporation of the block operations to the FPM programming model is also rather straightforward. The differences to the ordinary future requests are that the block requests have to be done for arrays and they require the specification of the length of the array to be transferred. The syntax of the blocked future request is thus

**future**(k) local_array[L_index] := shared_array[S_index]; (5-8)

where *local_array* and *shared_array* have at least $L\_index+k+1$ and $S\_index+k+1$ elements, respectively. Analogously, the syntax of the blocked shared memory write request is

**fwrite**(k) shared_array[S_index] := local_array[L_index]; (5-9)

with the same restrictions on the array lengths as with the *bfuture*-reference.

### 5.2.3 Synchronization

The most robust synchronization primitive of the FPM model is the synchronization of the whole machine as defined in the F-PRAM model. The synchronization of the whole machine requires every processor to participate in the synchronization, and provides us a solid basis of synchrony for all processors and shared memory references. The strict requirement of every processor having to participate is, however, the problem of this primitive. If one of the processors does not participate, but skips the synchronization, all other processors will wait forever waiting for the one that skipped the synchronization. The requirement is not severe unless we need to design fault-tolerant algorithms. In case of a processor fault, the synchronization would dead-lock. We have, however, left the fault-tolerance issues out of this thesis.

The syntax of the whole machine synchronization primitive of the FPM programming model is simply a statement

**synchronize**; (5-10)

which calls the synchronization routine. The processors reach the statement independently, depending on the synchrony of the execution. While executing the synchronization routine, the processors wait until every processor has reached the synchronization, distribute the knowledge of that, and continue their executions from the next statement. Furthermore, the synchronization guarantees the synchrony of the shared memory references. In other words, all shared memory writes that were issued before a synchronization will be fulfilled before any references that are issued after the synchronization.

We shall not define any other synchronization primitives than the barrier synchronization. Instead, we shall discuss some programming facilities to synchronize a smaller set of processors or data accesses without synchronizing all processors. Since we use the SPMD approach, the processors are executing the same task in approximately the same phase. Thus, the processors can maintain one or more counters in the shared memory which represent the phase of computation they are executing at any given moment. If the processors maintain equivalent counters, they can detect the phase of another processor by reading its counter. Similarly, any data can be tagged with a counter, which then represents the stage it was last written. We have to be careful, however, to be sure that the value and the counter get written by the same processor. Another possibility to synchronize a pair of processors is to use the synchronizing message passing Algorithm 4-1. Arbitrary sets of processors can be also synchronized using Algorithm 7-9 (page 160).

### 5.2.4 Read-only machine-characteristic variables

The use of the F-PRAM model is based on the use of machine-dependent parameters of the computer that we are using. We use the parameters to write programs that adapt themselves to work efficiently on different types of parallel computers. The parameters must be available for the algorithm designer in some form. The final values of the parameters cannot be available to the designer since they are machine dependent and, thus, not known at the stage of algorithm design. Hence, the FPM programming model includes a predefined set of read-only *machine-characteristic variables* that correspond to the set of the parameters of the F-PRAM model, and are available for the programmer. The machine-characteristic variables are read-only local variables. The values of the variables must naturally be equal in every processor since they are often used to evaluate the ranges of parallel loops. The values of the variables will be assigned only at load time of the program.

Some of the more complex features will be presented as simple functions instead of variables.

The set of the machine-characteristic variables is divided in primary and secondary variables exactly like the parameters of the F-PRAM model. The primary machine-characteristic variables are the number of processors $P$, the shared memory reference latency $L$, and the bandwidth inefficiency $B$. The definitions of the variables correspond to the definitions of the corresponding parameters of the F-PRAM model. The unit of $L$ and $B$ is "clock cycle." The secondary machine-characteristic variables are the block reference bandwidth inefficiency $B_B$, the processor shared memory reference overhead $B_P$, the single variable bandwidth $B_V$, and the synchronization cost $S$. The units of $B_V$ and $B_P$ are "simultaneous references" and "simple atomic operations," respectively. At this stage, we ignore the rest of parameters of the F-PRAM model to keep the FPM programming model reasonably uncomplicated. We shall discuss the modularized memory model with dedicated variables in more depth in Section 5.3.

The natural use of, e.g., the parameter $L$ is to adjust the number of local operations made before accessing the value of a future to ensure that the processor does not need to wait an unresolved future. The problem of this type of adjustments is the clock cycle as a time unit. If we trivially choose to make $L$ iterations, the actual time will be $C \times L$, where $C$ stands for the length of the body of the iteration in clock cycles. The goal was to use only about $L$ clock cycles. To avoid both unnecessarily long waits and unnecessarily long iterations, the correct number of iterations would be $L/C$. The programmer cannot know the length $C$ of a program block in clock cycles, but the compiler is able to calculate it at least if there is no inner iteration within the outer one. Thus, the FPM language includes a read-only variable *LBL* (Loop Body Length) that is assigned the estimated length (in clock cycles) of one iteration of the next *for-do* iteration. Then we can write iterations like

**for** i := 1 **to** L/LBL **do** (5-11)

to achieve a good balance between the latency and the computation time. In case of an *if-then-else* statement within the iteration, the estimation is done according to the longest branch of the conditional statement.

In addition to the above cost variables, the FPM model includes variable processor-id *PID*, which is distinct for each processor. As we defined in Subsection 5.2.1, the variables $P$ and *PID* describe the available number of processors and the processor-ids in the whole machine at the beginning of the execution. Functions $P(d)$ and $PID(d)$ describe the available number of processors and the processors-ids within the $d$th nested *par-do* statement. We shall represent the current depth of nested *par-do* statements with variable $D$. Additionally, we shall define that $P = P(D)$ and $PID = PID(D)$. Therefore, the number of processors in the whole machine is $P(0)$, and the original processor-id of a processor is $PID(0)$.

The use of the machine-characteristic variables corresponds to the use of the parameters of the F-PRAM model. The primary variables may be used in any program. A typical easy algorithm would use, e.g., $P$ and $L$. More sophisticated algorithms can use also $B$, $B_B$, and $S$. We can use the secondary variables when optimizing an algorithm for a given machine, or if they help us considerably to improve performance of a general purpose algorithm.

## 5.3 Options for the programming model

The previously presented FPM programming model is not the only possible programming model for the F-PRAM model of parallel computation. The FPM model is a nearly minimal model that is expressive enough to serve as a testbed for the F-PRAM model, but inadequate for real programming tasks. Consequently, in this section we shall discuss the possible modifications and options for the FPM model that still fit within the F-PRAM model. As these are only options, we shall not define the new features as accurately as the features that we included to the FPM model. Instead, we mostly discuss the features, give some rationale why we did not include them in the model, and give possibly some example syntax for inclusion in the programming model. Some of the following modifications replace some FPM primitives with alternative primitives that perform the same task. Others correspond to the options and secondary parameters of the F-PRAM model. Naturally, we could also define a totally new programming model if we wanted, but within this thesis, we shall hold to the FPM model. We shall first discuss some possible extensions and alternatives to the parallelism creation primitives in Subsection 5.3.1. In Subsection 5.3.2 we shall discuss some alternatives to the shared variable primitives of the FPM model.

### 5.3.1 Weighted par-do statements

Using the nested and possibly recursive *par-do* statements, we can present any parallelism pattern. More important question is, whether the processing power and processing needs are balanced in different parallel execution branches. The default definition of the *par-do* statement distributes the available processors evenly among the *par-do* statement range. If the following inner *par-do* statements have unbalanced processor needs, the even distribution is not optimal. Figure 5-1 presents examples of balanced and unbalanced nested *par-do* statements. The number in each square presents the number of processors available for the concerned program block. The left-hand program block has symmetric structure, and the processors get distributed evenly to the innermost iterations. Each of the iterations gets 0.5 processors. In other words, each of the processors gets 2 iterations to perform. The right-hand program has a slightly different inner parallel loop, the range of which depends on the loop variable of the outer parallel loop. Consequently, the processors get distributed unevenly among the innermost statements. A processor will not get anything to perform, three processors get one iteration each, two processors get 2 iterations each, and two processors get 4 iterations each to perform. The problem will be still worse in recursive parallel algorithms. In every level of a poorly designed recursive algorithm, half of the remaining processors may get practically nothing to execute, and the other half may get nearly everything. Finally, the last processor may get half of the work to do.

Our plain FPM programming model does not provide any easy solutions to the above problem. Using only the basic model, we need to flatten the nested *par-do* statements to a wider *par-do* statement and a serial iteration, and perform the indexing by hand. This approach will produce uglier programs because of the usually complex index calculations, which simulate the effect of several nested *par-do* statements. Fortunately we have not yet encountered a real-world problem of the above difficult nature.

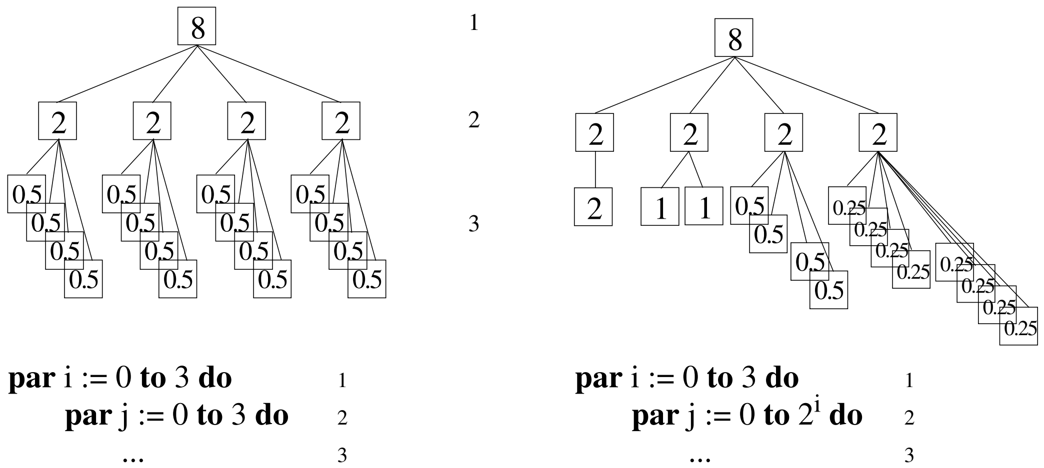


**Figure 5-1:** Balanced and unbalanced nested *par-do* statements.

A truly random or data-dependent work distribution cannot be balanced using any type of *par-do* statements, but the programmer should be able to characterize most usual predictable asymmetries. Via characterizing the asymmetry which is to occur in the following nested *par-do* statements, we could guide the compiler to assign the correct number of processors to each iteration of a parallel loop. Using statement

**par** i := low **to** high **do with** num_of_procs(i) (5-12)
...

we can assign each parallel iteration *num_of_procs*($i$) processors. The value *num_of_procs*($i$) is an expression depending on the value $i$. If the sum

$$P_W = \sum_{i = low}^{high} num\_of\_procs(i) \tag{5-13}$$

exceeds $P$, we shall run out of processors. The natural restriction $P_W \le P$ would solve the problem, but it would restrict flexibility and portability against the spirit of the FPM model. Our standard approach for running out of processors has been serializing the work for several processors by transforming the *par-do* statement to a *for..do* statement. Such an approach would require each processor to have knowledge of the $P_W$ and the inverse function of *num_of_procs*($i$), which can be difficult to arrange. Thus, we shall simply scale down the *num_of_procs*($i$) to the range where $P_W \approx P$. Consequently, the *par-do-with* statement (5-12) is implemented as

**par** i := low **to** high **do with** num_of_procs(i)*$P/P_W$ (5-14)
...

which scales down the number of required processors with reasonable accuracy. The problem of the above definition is that it requires the sum $P_W$ to be available for the processors. As the automatic computation of the sum would take too long, we have to require the programmer to include the sum in the *par-do..with* statement. Therefore, the syntax of the *par-do..with* statement will finally be

**par** i := low **to** high **do with** num_of_procs(i) **of** $P_W$ (5-15)
...

which is a slightly awkward form, which is the reason why we did not include it the basic FPM model. If we do not have a convenient function for $P_W$, we have to compute the value before the *par-do..with* statement. The summing will take some time, but usually it is not needed at all. Our previous example of Figure 5-1 can be balanced by writing it in form

**par** i := 0 **to** 3 **do with** $2^i$ **of** 15 (5-16)
**par** j := 0 **to** $2^i$ **do**
...

which would assign all subtasks the correct number of processors.

## 5.3.2 Alternative shared variable primitives

We defined the shared variables and the references to the shared variables of the FPM programming model as straightforward as possible. The F-PRAM model does, however, allow more sophisticated primitives for using the shared memory. Furthermore, the options for the F-PRAM model defined in Section 4.6 allow more possibilities for the programming model. In this subsection we shall discuss some possible options for the shared variables of the FPM model.

### Read-modify-write

We discussed the usefulness of an atomic read-modify-write operation in Subsection 4.6.4. From the viewpoint of a processor, the behavior of the operation is close to the behavior of a future. Consequently, we shall define the syntax of the operation similarly as the future. The difference is that besides the name of the variable, we need also to define the modifying function. To keep the reference straightforward, we shall allow only the basic operations: addition and multiplication by a value. The syntax will be

**modfuture** local_var := shared_var *oper* expr; (5-17)

where *oper* is either an addition or a multiplication, and *expr* is a locally evaluable expression. At the execution of the statement, the *expr* will be evaluated first locally, after which it, the operation, the address of the *shared_var*, and the return address *local_var* are sent to the shared memory. In the shared memory the value of the location *shared_var* is read and sent back to the *local_var* of the requesting processor. Within the same atomic operation also the operation

shared_var := shared_var *oper* expr (5-18)

is executed by the shared memory. The original value of the *shared_var* will get assigned to the *local_var* within expected time *L*. If necessary, we can naturally allow also other operations, possibly even our own functions. If we allow functions to be included, we have to somehow charge the cost of the operation. Because the cost occurs at some other processor,[51] we need to either amortize the costs, or define the data distribution, and charge the cost from the known processor.

### Replace future

In Subsection 4.6.4 we mentioned replace primitive as an easier-to-implement alternative for the read-modify-write. In addition to the possibly easier implementation, the replace is also easy to define. The informal definition is the combination of the *fwrite* and *future* operations. The syntax would thus be, e.g.,

**replacefuture** local_var :=: shared_var; (5-19)

which differs from the *future* by the fact that the value of *local_var* will be written to *shared_var* at the same indivisible operation when the value of the *shared_var* is read to the return packet.

### Data distribution directives

Since multiport, independently word-wise accessible memories have not been built yet, we have to expect that the shared memory is more or less modularized. In Subsection 4.3.2 we defined the secondary parameters memory module bandwidth $B_M$, and the number of memory modules $M$ for the F-PRAM model. Furthermore we stated that we need to be able to define the data distribution scheme to be able to efficiently use the modularized memory model. The parameters $B_M$ and $M$ can be easily realized in the FPM programming model by defining them as machine-characteristic variables. The data distribution scheme is much more complex to define in a form which would be both expressive and usable. The High Performance Fortran standard proposal has a rather good system, which relies on virtual processors, templates and alignment of data according to other data or templates [47]. We shall settle for a more simple model of only determining the distribution scheme of each shared array separately.

The goal of the data distribution is to distribute both the data and especially the references evenly among the memory modules. The distribution scheme of the shared scalar variables is not very important since they are rarely used concurrently.[52] For example, we can use random or cyclic distribution for the set of shared scalar variables. More important is the distribution of arrays since the different elements of an array are usually referenced concurrently by different processors. Our goal is to distribute the elements so that the concurrent references would fall on different modules of the shared memory. The pattern of references is naturally very algorithm-dependent and possibly hard to detect from the source code of the program. Furthermore, an array can be referenced by several different patterns during the program execution. Hence, the compiler is not usually able to optimize

51. Complex read-modify-write operations can be accomplished only if the shared memory is distributed among the processing nodes.

52. If scalar shared variables are used concurrently, they are bound to be hot spots.

the distribution without our help. Our help is the directives that guide the distribution of each shared array of the program. Naturally, finding the optimal distribution can be difficult for us also. Especially difficult are the arrays which are used differently in different parts of an algorithm. In such a case there might not be a solution which would avoid collisions totally, but we need to find the distribution which minimizes the collisions in the most time-critical parts of the program.

We shall define cyclic, block, and random distributions. The distribution directives must be included in the array declarations, more accurately in the dimension indices of the arrays. The syntax of an array declaration with distribution directives is

array_name : **array** [*distr1* dim1, *distr2* dim2, ... ] **of** *type*; (5-20)

where *type* is the type of the elements and *distr1* is the distribution scheme of the first dimension of size *dim1*. The distribution of each dimension can be defined independently. The possible schemes are:

*cyclic*
: Modulo distribution. Element $k$ gets assigned to memory module $k \bmod M$.

*cyclic(bl)*
: Blocked modulo distribution. *bl* successive elements get assigned to the same module, the next *bl* successive elements to the next module, and so on. *cyclic(1)* equals to *cyclic*.

*block*
: Division distribution. The first $\lceil N/M \rceil$ elements get assigned to module 0, the next to the module 1, and the last to the module min($N$, $M$–1). The directive *block* equals to *cyclic*($\lceil N/M \rceil$)

*block(blocks)*
: Division distribution to the first *blocks* modules of the shared memory.

*random*
: Distribution using some reasonably random and $O(1)$ time computable hash-function.

Even if the list contains some redundancy, it is more illustrative than the minimal two primitives.

Since the F-PRAM model does not recognize any nearness or locality within the shared memory, the distribution is only meant for avoidance of collisions. If the transfers from the "local" or "near" module of the shared memory were faster than from far modules, the optimal distribution would be still more important, and more difficult to find and define. Also, the block transfers of successive data elements, require blocked data distribution with correct boundaries to be able to provide optimal data transfer rates. Cyclic or random distribution would make block transfers more difficult in a truly distributed shared memory.

# Chapter 6

# Experimental tools for testing the F-PRAM model

The goal of the F-PRAM model and this thesis is to model and study the impacts of different properties of parallel machines on different algorithms. The problem is that the set of parameters is large, and the range of each parameter is even larger. We cannot test the impacts of all these parameters on existing parallel computers because there do not exist computers with all these properties. Using one multimillion dollar parallel computer, we can study the impact of the number of processors, and compare different algorithms. Having two different parallel computers with different latencies, we could study the impact of latency, except that the computers would be different also in dozens of other ways, so that the results would not be valid anyway. Using a set of workstations, and testing the same algorithms with different Ethernet HUBs and/or switches, we could get a bit more valuable information a bit cheaper. In general, however, there is no economic way to separately measure the impact of different parallel machine properties in real parallel computers.

We cannot measure the impacts of the parameters on real parallel computers. Instead, we can measure them on a simulator. There are often demonstration/development simulators of actual parallel computers provided by the manufacturers at the time of their development. A more general machine simulator to simulate hardware using existing software and operating systems is the SimOS made at Stanford [89]. Modeling and simulating whole machines in unnecessarily complex for algorithm research. Instead, we prefer simulation of the models of parallel computations. We are aware of a few PRAM simulators/emulators, e.g., [18, 53, 81] and a few general architecture simulators, e.g., [19, 73]. Additionally there are a lot of proprietary tools written to support a more accurate analysis of algorithms. The only simulator of the newer parameterized abstract models that we are aware of is the recent BSPlab developed at NTNU, Norway [101]. An emulator of a parameterized model allows us to study the effects of the parameters, i.e., the parallel machine properties. For this purpose we have constructed an emulator system to do the measurements in a simplified model of parallel computers. Here we have to emphasize that the system does not simulate any physical properties of any parallel computer. Instead, it emulates a theoretical parallel computer. In other words, it executes a separately defined machine language using some abstractions of different parallel machine components. As the emulator is a software product, we can fully configure the features, i.e., the behavior, of the emulator. In particular, we can freely choose the values for parameters $P$, $B$, $L$, etc. for each run of the emulator.

In addition to the total configurability, the benefits of an emulator include the absence of implementation specific anomalies, such as superlinear speedups due to the cache or memory increase, or inability to fit the whole execution in a single processor. A shortage of the emulation approach is that we cannot use as large data sets as we would using real hardware. In spite of the rather slow software emulation, we have, however, been able to run big enough data sets to show up the characteristics of all algorithms we have written this far.

In Section 6.1 we shall present the features and the implementation of our experimental F-PRAM emulator. In Section 6.2 we shall present the implementation of the experimental FPM compiler for the experimental F-PRAM emulator. Besides the actual compiler and emulator system, in Section 6.3 we briefly present the measurement system used to generate the graphs to be presented in Chapter 7.

## 6.1 An F-PRAM emulator

Our concept of emulating the F-PRAM model consists of a definition of a theoretical machine fulfilling the F-PRAM model requirements and an emulator program to execute the programs written for the theoretical parallel machine. We separate the definition and the implementation of this emulator to avoid a definition by an example, i.e., to be able to simulate the theoretical machine model by later emulators or real parallel computers as well. Even if the definition of the machine is similar to the definition of the whole F-PRAM model, we shall give it here again because of the additional level of details needed for a machine model to be emulated automatically. Also, we shall repeat some reasonings about some features to support the decisions we have made in designing the emulator system.

### 6.1.1 A theoretical machine model for the F-PRAM model

The skeleton of the definition of the emulated machine model is based on the definition of the F-PRAM model itself. Especially the components of the parallel components are equal to the components of the F-PRAM presented in Chapter 4. Here we shall define the function of the components more accurately to form a "working parallel computer."

**Definition 6.1:** A theoretical F-PRAM machine consists of

- $P$ processing nodes (RAM, Random Access Machine), each having
  - a sequential processor
  - local memory accessed wordwise
  - local copy of the program
  - access to the interconnection network and to the synchronization network
  - an input and an output stream
- a shared memory accessed wordwise via the interconnection network

- an interconnection network with a connection to every processor and to the shared memory
- a synchronization network with a connection to every processor

### RAMs in the F-PRAM

Each processing node of the F-PRAM is an extended version of a standard von Neumann style random access machine (RAM) [91] found in most CS textbooks. The extensions include the facilities to access the shared memory and the synchronization network. The processor consists of

- an accumulator and a set of other general purpose registers
- a set of special read-only registers to hold the values of the F-PRAM parameters
- a program counter (PC)
- a processor index (PID)

The processor executes simple machine language (ML[53]) instructions one at a time sequentially. The instruction set includes

- arithmetic instructions using accumulator and/or one other register as operands and storing the result to the accumulator
- *read* and *write* instructions for transferring data from/to the local memory to/from the accumulator
- branch instructions (unconditional and conditional ones based on the accumulator)
- *future* instruction for launching a future-request to transfer data from the shared memory to the local memory
- *fwrite* instruction for initiating a transfer of data from the accumulator to the shared memory
- *synchronize* instruction to participate in a global synchronization
- *halt* instruction to stop the execution

By default the execution of any standard instruction takes one clock cycle. The execution of the *synchronize* instruction takes $S$ clock cycles after all processors have started to execute it. The execution of *future* and *fwrite* instructions take $B_P$ clock cycles.[54]

### Shared memory

The shared memory of the F-PRAM machine is a continuous wordwise indexed random access memory. The shared memory system handles the memory references issued by the processors and delivered by the interconnection network. The order and the volume of the references is determined by the interconnection network, i.e., the shared memory handles all the references it gets. The fwrite references are obviously written to the correct mem-

53. In this thesis we do not refer to the functional language ML.

54. By default, $B_P = 1$.

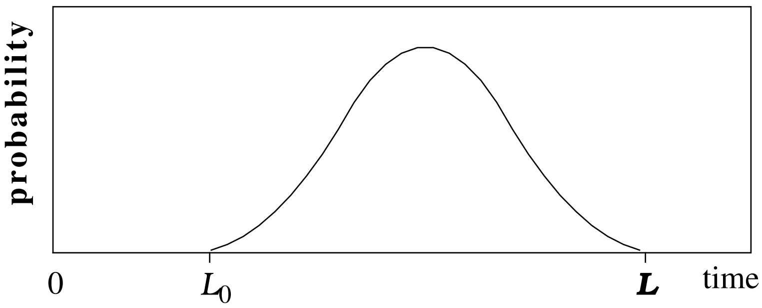


**Figure 6-1:** A possible latency distribution.

ory location in the order they are received. The future references are filled by the value of the correct memory location and sent back to the interconnection network.

The definition above declared that the shared memory is continuously indexed, but does not require it to be continuous itself. In other words, the memory can consist of one or more separate memory banks as long as global indexing of the memory locations is continuous. In practice, the global memory locations are hashed to the several banks using a properly chosen hash function. If the memory consists of several banks, the interconnection network must have separate connections to each of them. Moreover, the interconnection network has to be able to send a shared memory reference for the correct memory bank. In practice, this can be implemented either in processing node or in the interconnection network. If the processing nodes transform the global address to the number of a bank and the address within the bank, the interconnection network just delivers the request to the correct bank. On the other hand, the routing nodes of the interconnection network may be able to decide the correct destination using only the memory address. The latter approach is efficient only if the decisions can be done in each routing node based on one or few bits, or otherwise as easily. Examples of this are all hypercubic networks. If we use proper hashing of the memory addresses, the unhashing should be done by the processing node. Recalculating a strong randomizing hash function in every routing node would be a waste of resources since it can be done in the network connection of the processing node, and attached to the message.

### Interconnection network

The interconnection network delivers the shared memory references from the processors to the shared memory system, and, in case of future requests, returns the packets back to the processing node. To keep the definition simple and portable, we do not require any actual topology of the network. Instead, we use the parameters $B$ and $L$ of the F-PRAM model to set the requirements on the network. In an unsaturated interconnection network, a future request shall be server within $L$ clock cycles with high probability. For example, the times required by a set of future references could distribute mostly between $L/2$ and $L$. Figure 6-1 presents a possible graph of latency distribution in a non-collision network with topology of a 3-dimensional mesh. Suppose that the latency of serving request $i$ is of the form

$$L_i = L_0 + 2 \times C \times distance(\text{source node, destination node}), \tag{6-1}$$

where $L_0$ is a base latency, and $C$ is the time used in one intermediate routing node.[55] The base latency typically consists of the time used by the network connection of the processing node, and the time used by the memory module. If the references distribute evenly among all processing node within the cube, we get a bell curve of latencies as seen in Figure 6-1. Now we can set the parameter $L$ to the point where we can assume all memory references to be served, i.e., $L = L_0 + 2C\lceil \sqrt[3]{P} \rceil$.

**Synchronization network**

In the theoretical machine model, we require the processors to have a separate instruction *synchronize*, which causes them to freeze until the synchronization network releases them to continue their execution. The synchronization network waits until all processors have executed the synchronize instruction, and after that it sends the releasing signal to the processors within $S$ clock cycles. All processors need not to be released simultaneously. This definition does not allow any submachine synchronizations or several concurrent pending synchronization waves.

### 6.1.2 An experimental implementation

The above definition of a theoretical machine model is made to be easily implementable using a software emulator system. In this section we shall present the experimental implementation used in the experiments described in Chapter 7. The current emulator system is a result of several consecutive stages. The first implementation was a PRAM emulator made by Pasi Hämäläinen in 1991-1992 [53]. When the F-PRAM was defined in 1995, we also redefined the emulator, and Jukka Veräjäntausta implemented the F-PRAM emulator in 1996 [105]. The current form was completed in 1997 when the author added the floating point arithmetics and some new instructions to the emulator system. The next stage in this branch of the research would be to rewrite the interconnection network and shared memory modules of the emulator.

As the definition in the previous subsection stated, the emulator accepts an F-PRAM assembler language program and executes it using $P$ processors. In order to be able to execute an assembler program to be executed efficiently, we first have to compile it to a machine language program. In this stage the possible macros are expanded, labelled jumps are interpreted to addresses, and the instructions are coded to be efficiently executed.

In addition to the ML program, the emulator takes the values of the F-PRAM parameters, and a possible input file. The F-PRAM assembler can be written manually, but in most cases we want to use a high level language and a compiler, which we present in the next section. In Figure 6-2 we can see the whole system, and the stages we need to execute a program. The FPM source code, input data, and the F-PRAM parameters are given by the programmer or user (or generated, as we shall see later in Section 6.3).

55. More accurately, the delay (time) in one node plus the time in the previous wire.

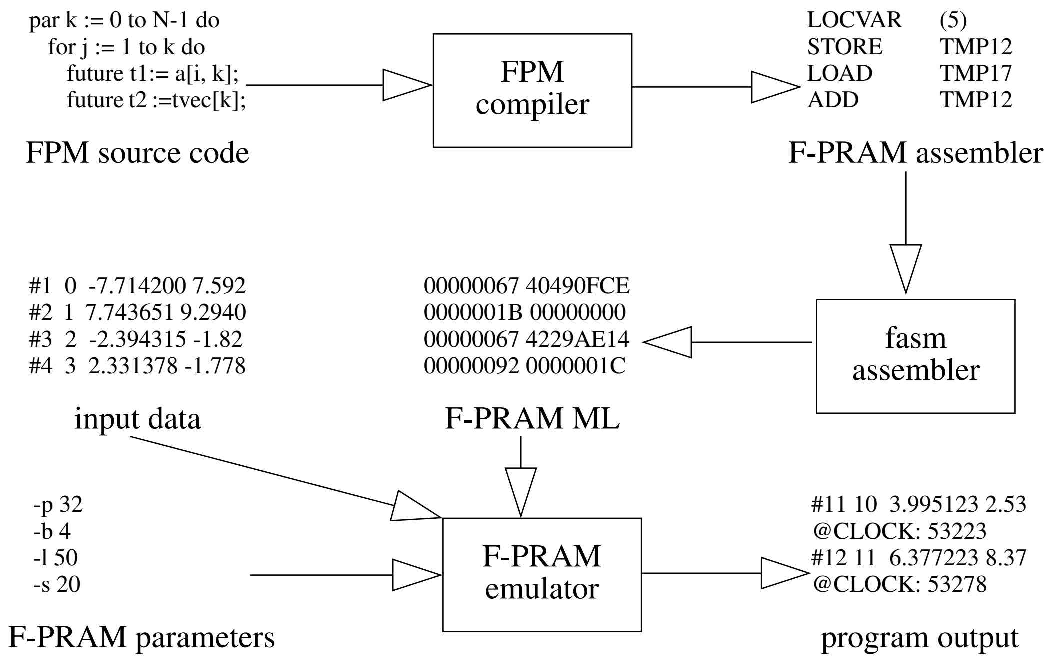


**Figure 6-2:** The stages of the F-PRAM emulator system.

**Principle of operation**

The emulator is designed to be executed in a sequential computer. The time unit of the emulation is clock cycle as defined above. Consequently, the emulator accomplishes all the operations needed to be done in one clock cycle in some order, and proceeds to the next clock cycle. During each clock cycle all of the components of the F-PRAM machine need some attention. For each processor the emulator executes one clock cycle. Usually this means executing one instruction and increasing the program counter by one.[56] As some instructions can take more than one clock cycle, the execution of the instruction may be continued on the subsequent clock cycles. Within the interconnection network, all references are forwarded one step. In the shared memory module, all references that arrived are served and possibly sent back to the network. If there is a synchronization pending, the emulator checks whether all processors have started to execute the synchronize instruction. Algorithm 6-1 presents the sequence of emulation more formally.

The cycle-by-cycle emulation of a simplified instruction set was chosen for portability and ease of analysis. This approach to emulation is, however, quite slow compared to the speed of real microprocessors. The host computer executes hundreds of instructions for each emulated instruction. The other shortcoming (and benefit in some cases) is the lack of some real-world features, especially caches. Another approach for emulation would be using a Unix thread for each processor, and having separate handlers to coordinate the threads and the parallelism-related requests [19]. Compared to the cycle-by-cycle emulation, the thread-based approach provides much better speed and the some of the impacts of real processor features such as the caches and slow memory.[57] Moreover, we

56. Or setting the program counter to the destination of a branch instruction.

```
initialize emulator
    while not all processors halted do
        for all processors do
            read next instruction
            execute instruction
        advance the queue of the pending shared memory references
        serve the memory references that reached the shared memory
        advance the queue of the returning shared memory references
        store the returned results of shared memory references to local memories
        clock := clock + 1
print output streams
```

**Algorithm 6-1:** Emulation of the F-PRAM.

can use existing compilers to produce to sequential code. The disadvantages are that the system would be less portable, and much more difficult to analyse.

### Interconnection network

Within the emulator, the interconnection network models the possibly poor connection between the processing nodes and the shared memory. A faithful implementation would be to build a graph of routing nodes between the processing nodes and the shared memory.[58] During the emulation we would then simulate the routing of the packets in the network cycle by cycle.[59] To study different networks, we would have to implement the simulation of all topologies. Using this implementation the exact meaning of parameters $B$ and $L$ would have to be measured and analysed carefully for each network. Our initial goal was, however, to be able to set the parameters accurately. To achieve this, we model the interconnection network with a queue of width $P/B$. More formally, the width $w$ of the queue is

$$w = \begin{cases} \lceil P/B \rceil & \text{if } P \geq B \\ \dfrac{1}{\lfloor B/P \rfloor} & \text{if } P < B \end{cases} . \tag{6-2}$$

When $w < 1$, the queue only includes one net load position at every $\frac{1}{w}$th position of the queue (clock cycle). Figure 6-3 presents an example of both types of the queue. In the opposite direction, i.e., from the shared memory to the processing node, there is also a similar queue.

57. The desired effects of, e.g., caches in emulation do not, however, necessarily reflect the effects in real parallel computers as all the threads exploit the same cache.
58. Or only between the processing node/shared memory block pairs, if we would emulate virtual shared memory.
59. We could also model easily a slower or faster network by advancing the packet only every $n$th clock cycle or by advancing the packets several steps on each clock cycle.

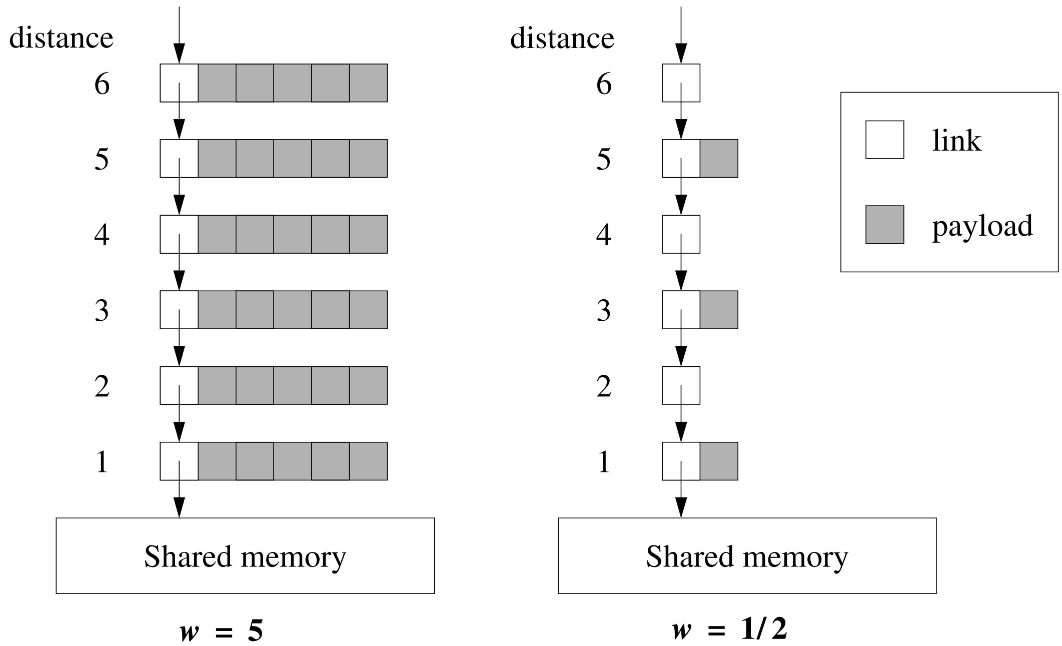


**Figure 6-3:** The interconnection network implementation of the current emulator.

When inserting a new shared memory reference into the queue, the emulator models latency $L$ by locating the new packet of the memory reference at distance $f(L)$ to the queue. If the position $f(L)$ is already full (by the rule of Formula (6-2)), the packet is assigned to the next free position in the queue. If the network is severely overloaded, the length of the queue grows long. The function $f$ can be chosen to be any function as long as

$$2 \times f(L) \leq L \text{ with high probability,} \tag{6-3}$$

where the factor 2 comes from the fact that the memory references should also be routed back to the referencing processing node within the $L$ clock cycles. Of course, if the network is saturated by too many references, the Inequality (6-3) does not hold. Our current implementation provides a default random function with Gaussian distribution centered at $\frac{2}{5}L$ with deviation $\sigma = \frac{1}{5}L$. Although neither this function, nor the queuing model do not represent any real network, the latency distribution has very similar properties with, e.g., the distribution of a 3-dimensional mesh of Figure 6-1.

### Parallel input and output

In a theoretical computation model, such as the PRAM model, it is generally accepted that we ignore the input and the output of an algorithm. Usually the input is assumed to reside in the shared memory before the execution and the results are assumed to be left in the shared memory after the execution. In our previous PRAM emulator [53] this was the solution. It only supported separate sequential program segments to do the actual I/O before and after the actual parallel computation. Within the F-PRAM model we want to examine also the costs of the I/O. Since the linear time needed in sequential I/O can destroy the efficiency of a fast parallel algorithm, we need parallel I/O. Furthermore, since some data might be needed by one processor only, we can reduce the possibly expensive shared memory usage by inputting the data to the processing nodes directly. Similarly, the final output generated by each processor needs not be transferred via the shared memory at all.

The definition of the parallel I/O follows the earlier definition of the F-PRAM model, i.e., each processor has an input and an output stream of its own. The instruction set of the processors includes calls to external routines that either read a value from the input stream to the accumulator, or write the value from the accumulator to the output stream. There are separate versions of the routine for input and output of integers, real numbers and characters (which are handled as integers in the emulator). In practice, the streams are stored in files, where there is one or more lines of input values for each processor. Each line is tagged with the corresponding processor-id.

#### Performance counters

The goal of the emulator system is to measure the performance of our algorithms in different machine configurations. The performance is measured as used time for a given operation. In an emulator, there is no sense in concept wall clock time since the emulator can be run in any environment, and it is always sequential no matter what set of parameters we assign to it. Consequently, the only usable time unit is the clock cycle. The emulator system maintains a global clock, and we need to access the values of the clock to monitor the time usage of our algorithms. Within the F-PRAM assembler language, we can insert a directive @*CLOCK* into any point of the code. During the execution, if a processor executes an instruction with the directive, the emulator prints the value of the global clock to the input stream of the processor. In addition to the @*CLOCK* directive, the current implementation also includes the directive @*LASTMEMINDEX* to print the amount of shared memory used within the computation.

## 6.2 Implementation of an FPM compiler for the F-PRAM emulator

In Chapter 5 we defined a new programming model for the F-PRAM model. To be able to exploit the programming model and the emulator system, we have implemented a compiler from the FPM programming model to the F-PRAM emulator. The compiler has been developed together with the emulator. The first version of the compiler was a parallel Modula-2 compiler for PRAM [60]. Later, we modified the language and the compiler to support the F-PRAM model of shared memory access. In this section we shall present the most important parallelism-related compilation techniques used in the FPM compiler.

### 6.2.1 Parallelism handling

As stated in the definition of the F-PRAM emulator, all processors execute the same program code independently. From the programmer's point of view, it might be a bit difficult. Therefore the programming model presents a single parallel program, which is then compiled to a single program to be executed by every program. In other words, the compiler takes care of the branching of the processors. Consequently, we can claim our language to be a Single Program Multiple Data (SPMD) model of programming, although the acronym SPMD is often used in several different meanings.

### Processor-IDs and the number of processors

The emulator system generates the processors at the initialization of the emulator, and after that the system has a fixed amount of processors, as the model assumes. The processors have access to the parameter *P*. The emulator also assigns each processor a distinct index, processor-ID (*PID*), within the range 1..*P*.

The compiler uses its own numbering for the processors to be able to more easily and more dynamically assign each processor different threads[60] of execution. During a computation, each processor maintains independently the *effective number of processors* in the same thread of execution (*P*) and its own *effective index* (range 0..*P*–1) within the thread (*PID*). At the beginning of a computation, all processors form a single thread of execution, and, thus, have *PID*s in the range 0..*P*–1. Within each par-do statement, the sets of processors are divided to a number of threads distinguished by the iteration variable. The processors maintain a stack (actually a table) of the *PID*s and *P*s of the nested *par-do* statements. Therefore, when exiting a *par-do* statement, the processors restore the earlier *PID*s and *P*s. Moreover, the processors can reference the *PID*s and *P*s of outer levels of par-do statements by using references *PID(level)* and *P(level)*. Within the rest of this section, *P* and *PID* refer to the effective *P* and *PID*, respectively.

### Execution threads

Within an execution thread, all processors execute the same program to keep the values of the local variables correct, usually identical. Only input/output statements and *fwrite* statements are executed by only the processors having effective *PID* 0. This is ensured by an additional *JPOS* instruction before the statement. Using the explicit *all* directive in a statement, the programmer can force all processors to do the operation.This can be used, e.g., outside any *par-do* statements to write a local value to a position of a shared array

**fwrite all** max_array[PID] := local_max; (6-4)

as in maximum-finding routine of Algorithm 7-8.

### *Par-do* statement

The standard form of the *par-do* statement is

**par** j := a **to** b **do** (6-5)
    *statements*;
**end**;

60. We defined thread in Subsection 5.2.1 as the processors executing a *par-do* statement with the same value of the iterations variable.

where $j$ is the iteration variable and $b-a+1$ is the *range* of the *par-do* statement. The statements are executed once for each value of the range. If

$$P \leq \mathrm{b} - \mathrm{a} + 1, \tag{6-6}$$

i.e., there are no more processors than the width of the range of the *par-do* statement, the statement is executed as

```
for j := a + PID to b by P do                    (6-7)
    PID := 0; P := 1;
    statements;
end;
```

that is, each processor executes the statements once for a value of the iteration variable. Since each thread is executed by only one processor, each processor will have the new *PID* 0, and the number of processors $P$ will be 1 in every thread. This is a rather coarse but very efficient method since the processors need not communicate with each other before, during, or after the *par-do* statement. Concerning the coarseness, dividing the work of 3 statements among 2 processors to be executed faster than sequentially executing 2 statements is difficult or impossible in the general case. Consequently the difference 1 is the best we can do in balancing the work between the processors.

The opposite case of Inequality (6-6), the case when the number of processors is greater than the width of the range in the *par-do* statement is a bit more difficult to handle. Not all of the processors are required to execute the statements in parallel. Using the solution of (6-7), the extra processors would skip the whole *par-do* statement. This would not be a valid behavior if the statements include one or more inner *par-do* statements. Because the inner *par-do* statements provide more parallelism, they are able to exploit more processors. Consequently, all the processors must execute all the statements to be able to participate in the possible inner *par-do* statements. Each of the processors will get assigned a value of the iteration variable. The processors with the same value of the iteration variable make up an execution thread. Thus, the number of threads will be $b-a+1$. Within an execution thread, one processor will receive 0 as a new *PID*, and the possible other processors will receive positive integers as their new *PID*s. Within each thread, the new number of processors is evaluated independently. Especially, if the old number of processors is not divisible by the width of the range, different threads will have different new numbers of processors. The difference will be at most one. Figure 6-4 presents an example of 8 processors and two nested *par-do* statements. The first stage divides the processors into 3 threads, each having 3 or 2 processors. The second *par-do* statement further divides each thread to 4 subthreads, each having only one processor, and some processors having to execute two threads. Figure 6-5 presents the changes in the local variables of the processors in the same situation. At the last row of the figure we notice that some of the processors execute the innermost statements for two different values of the iteration variable $j$.

As the figures suggest, we use blocked distribution of processors for the threads. More exactly, for a new thread $i$ we assign a set of processors having their old *PID*s within range

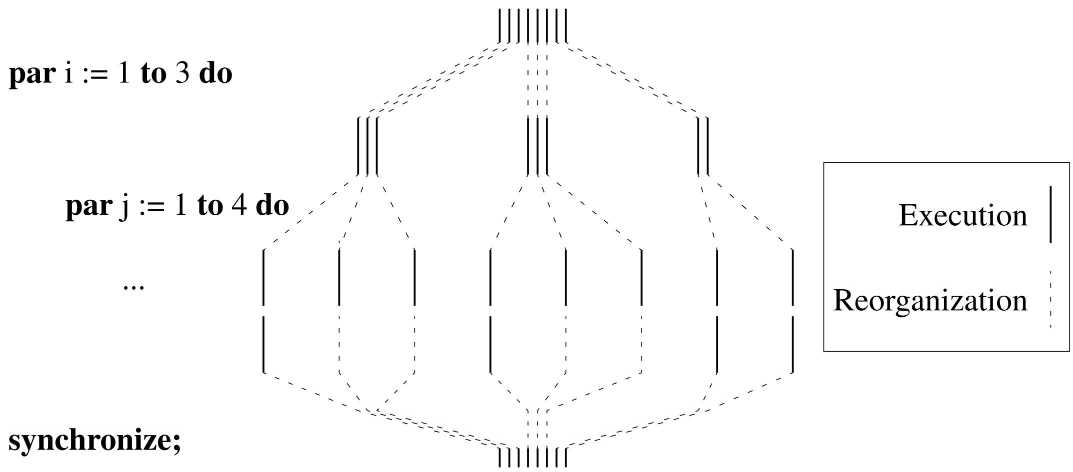


**Figure 6-4:** Processor usage in two nested *par-do* statements.

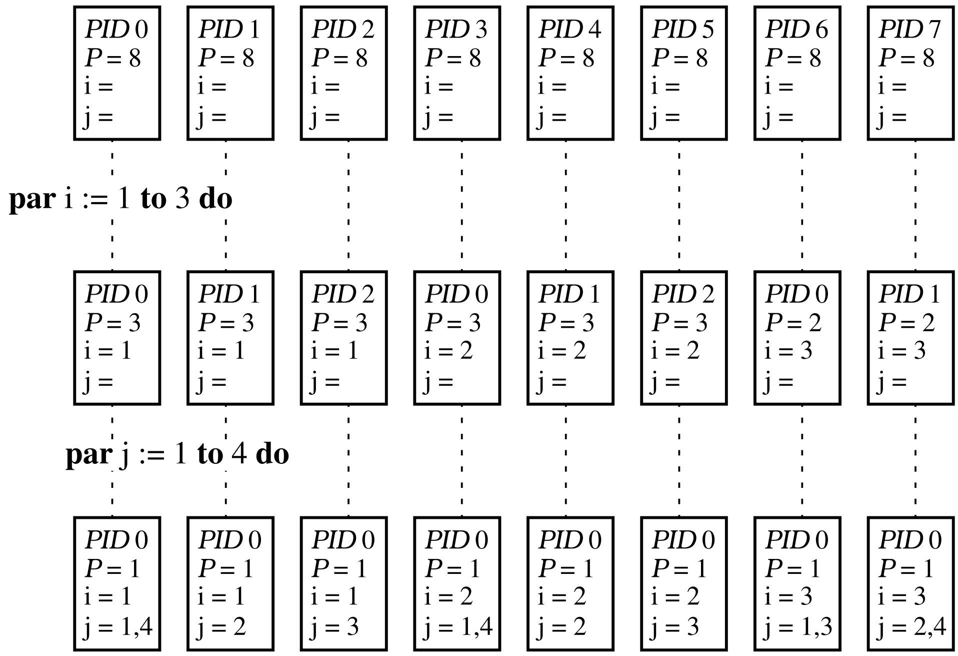


**Figure 6-5:** The number of processors, PIDs, and iteration variables in nested *par-do* statements.

$$i\times(P_{old}/\,range) \;..\; (i+1)\times(P_{old}/\,range) - 1, \tag{6-8}$$

i.e., a set of $P_{new} = P_{old}/\,range$ processors for each thread. Each processor will independently calculate a new *PID* within the new thread using formula

$$PID_{new} = PID_{old} - i\times(P_{old}/\,range). \tag{6-9}$$

Also, we have to consider also the probable remainder in the division $P_{old}/\,range$ so that all of the processors get used as evenly as possible. Combining and refining Formulas

```
1  range := b - a + 1;
2  if range ≥ P then
3      j := a + PID;
4      PID := 0;
5      P := 1;
6  else
7      bndr := (P_old mod range) * ⌈P_old / range⌉;
8      if PID ≥ bndr then
9          P := ⌊P_old / range⌋;
10         j := ⌊(P_old - bndr) / P⌋ + P_old mod range + a;
11         PID := (PID - bndr) mod P;
12     else
13         P := ⌈P_old / range⌉;
14         j := ⌊PID / P⌋ + a;
15         PID := PID mod P;
16 while j < P do
17     statements;
18     j := j + P;
```

**Algorithm 6-2:** Implementation of the *par-do* statement (6-5).

(6-7), (6-8), and (6-9), we can express the *par-do* statements as Algorithm 6-2. In the situation, where there are more processors than available parallelism (lines 6-15), the algorithm handles the processors separately depending on whether they will belong to a thread with $\lfloor P_{old} / range \rfloor$ or $\lceil P_{old} / range \rceil$[61] processors. In the former case, the new $P$ and $PID$ will be calculated with respect to the boundary of the two sets of processors. Implementation of the algorithm in machine language is rather straightforward, although the optimized machine language execution requires some merging and rearranging of some of the expressions. In addition to the presented algorithm, the processors also have to push the old values of $P$ and $PID$ to the parallelism stack, and pop them back after the statements. In a compiled program, starting and finishing a *par-do* statement takes about 45 clock cycles if $P < range$, otherwise it takes approximately 65-80 clock cycles, depending on which branch of the if statement of line 8 of Algorithm 6-2 the processor takes.

The most notable feature of Algorithm 6-2 is that every processor independently calculates the new values of $P$, $PID$, and the iteration variable using only the old values of $P$ and $PID$. Consequently, no interprocessor communication is needed during single nor nested *par-do* statements. Furthermore, the statements can be executed fully asynchronously. At the extreme, a processor can complete the statements before another processor enters the statements.

### Miscellaneous F-PRAM specific features

All the direct memory references made by the compiler are local. Following the F-PRAM model, the language specification requires that all shared variable references are made by

61. The use of the floor and roof operations are avoided in practice using the knowledge of the groups of the processors.

the *future* and *fwrite* statements. For each shared variable reference, the compiler generates both local and shared addresses (values) and issues the corresponding ML instruction as the machine model requires.

Synchronization statement of the FPM language is the barrier synchronization of the F-PRAM model. We have chosen to leave the responsibility to guarantee that every processor takes part to the synchronization to the programmer. Consequently, the compiler does not need to take any care besides inserting the synchronize instruction at the place of each synchronize statement.

One of the ideas of the F-PRAM is that the programmer can use the values of the F-PRAM parameters in his/her program to make the program to adapt to the used parallel computer. For each F-PRAM parameter there is a predefined read-only variable in the language. The variables can be used to refer the properties of the computer. The references of the variables are compiled to references of the corresponding virtual registers of the processors of F-PRAM emulator.

The parameters of the F-PRAM are given in clock cycles, which is not the best unit for the programmer. To be able to make, e.g., enough iterations to hide $L$, the programmer needs to know the length of the iteration also in clock cycles. For this purpose the compiler provides a read-only variable *LBL* (Loop Body Length) that is assigned an approximation of the length of the body statements of the next *for-do* iteration.

**Code generation**

In an experimental compiler, the code of which is to be emulated, the optimized code generation is not the most important issue. It does, however, have a constant effect when comparing the results of the emulated system to a real parallel computer. For example, if the body of an innermost iteration takes twice the clock cycles compared to a good compiler, the resulting code will probably tolerate twice the latency having the same slowdown.

The instruction set of the F-PRAM emulator has only arithmetic instructions getting input from the accumulator and/or another register and storing the result to the accumulator. Consequently, all addresses of (local) memory references have to be first calculated in the accumulator, and after that the value can be loaded into the accumulator. Only after that, the actual computation can be made. Expressions involving several operations or memory references require additional stores of every intermediate result to the spare registers or to the stack. Also, the stack push and pop can only be made through the accumulator. This accumulator-based architecture requires more instructions by approximately by factor of 2 compared to a more modern register-to-register or memory-register architecture. Moreover, most current microprocessors are 2-6-way superscalar, i.e., they can execute (issue) more than one instruction every clock cycle. On the other hand, the local memories are slow, and cache misses stall the processor for tens of clock cycles. These variations make the definition of a common time unit more difficult.

The compiler does its best in producing a reasonably efficient machine code for the accumulator-based architecture. Optionally the compiler produces optimized versions of most of the important structures. Moreover, the compiler includes an assembler optimizer that scans through the produced code, and combines and deletes instructions where possible. Part of the optimization responsibility has been left to the programmer. Especially, the compiler does not locate the variables to registers automatically since the variables may be used for futures, which work only with the local memory. Therefore, the programmer should tag the most important temporary variables with the keyword *register*.

As an example we study the innermost loop of matrix multiplication. The textbook version of the iteration is

**for** i := 0 **to** N-1 **do** (6-10)
c := c + A[x, i] * B[i, y];

where is $N$, $i$, $x$, and $y$ are local scalar variables, and $A$ and $B$ are matrices. On each iteration elements of two different matrices are multiplied and added to a local sum (which is later assigned to the result matrix). Additionally, the value of the iteration variable is increased and the termination criterion is tested. In an optimized implementation the references to the matrices are handled with pointers, both of which are increased at every iteration, and the termination test is also done using one of the pointers. Using the current compiler, the actual computation takes 8 clock cycles, increasing the pointers takes additional 2 clock cycles, the termination test takes 4 clock cycles, and the branches take 2 clock cycles. In total, each iteration takes 16 clock cycles. In Section 7.2 the measured cost of the whole matrix multiplication procedure also approaches the value $16N^3$ clock cycles. By hand optimization we could reduce the constant by 2 clock cycles by exploiting a more efficient termination test. Anyway, the compiler produces reasonably efficient code for the chosen architecture. In a more optimized architecture, the number of instructions would be smaller. For example, the Gnu C compiler [31] produces an iteration of length 8 instructions out of a comparable C program for the SPARC architecture. Using superscalar execution, the instructions might be executed in 4-7 clock cycles, if the vectors were in first level cache. In practice, however, the caches do not contain all the values, and, thus, at least one reference within the loop is slower. Vectorized code in a vector supercomputer should be able to execute the innermost loop at rate of 1 iteration per clock cycle, but these machines are at least as much more expensive as they are more efficient.

The reason for this comparison of the generated code is to be able compare our main time unit, clock cycle, in the emulator system and in the real parallel computers. In practice, it should suffice to see that the emulator needs two clock cycles to do the same work as an average microprocessors does in one instruction. When comparing with highly efficient systems, such as vector supercomputers, the factor is a bit larger. We need to find the approximate factor for each architecture to be able to make any direct comparisons.

## 6.3 Automated measurement system for the F-PRAM emulator system

The goal and purpose of the emulator system is to be able to study the impacts of different variable aspects of the execution. In practice, we want to know, for example, how much longer it takes to complete an algorithm in a computer with latency 1000 than in a computer with latency 500. Moreover, we want to know how the execution time rises as the latency rises from 1 to 10,000. Consequently, after we have checked that our algorithm produces the correct results, the most interesting result is the total execution time (in clock cycles) of the algorithm.[62] In this section we shall describe how we have used the tools presented in this chapter to test the model and to plot the graphs to be presented in Chapter 7.

**The problem**

The goal and purpose of the emulator system is to be able to study the impacts of different variables of the execution. The obvious variables are the F-PRAM parameters such as $P$ or $L$. In addition to these, the possible variables also include input size, the number of registers, possible variations in the algorithm, and compiler options. In total there is usually a set of approximately 10 variables to take into account when running a simulation. The variables may have a very large domain of possible (and interesting) values. For example, the latency may vary from 20 clock cycles (vector supercomputers) to $10^6$ clock cycles (LAN based network of workstations). Even if we used exponentially growing steps between the experiments (e.g., $L = 16, 32, 64, 128, ..., 2^{18}$ instead of 10, 20, 30, ...), the number of interesting sets of variables is vast. For example, if we have 5 variables with 5 interesting values each, and 5 variables with 20 interesting values, we would get $10^{10}$ different combinations of variables. Naturally, this is too many experiments to run. Besides, we cannot visualize the impact of more than two or three variables simultaneously. Consequently, in a rational set of tests, we vary 3 parameters with 10 values each, resulting 1000 distinct experiments. After the tests, we select two variables to be studied together, pick a couple of representative values of the third variable, and plot a set of curves or a 3D plane for each of the chosen values of the third variable.

In addition to the complexity induced by the sheer number of the test run to be made, the slight complexity of the emulator system as seen in Figure 6-2 makes the experiments tedious to perform manually. If we change the algorithm, we also have to recompile and reassemble it. If we use parallel I/O and change the number of processors, we may have to regenerate the input file. Consequently, we have perform up to five commands to get the execution running, and then extract the wanted results from the output of the execution.

**The solution**

As the base of our solution, we chose *files of test sets*. A test set is a set of all parameters. In a file of test sets, each row contains one test set. We generate these files using a separate awk script which accepts a *test set definition*. In a test set definition there is one parameter per line, and after each parameter there is one or more values for the parameter. The test set generator generates one test set for every combination of parameters. The files of test sets are fed to another awk script, which generates a suitable sh script to run all the needed stages of the emulation system to achieve the results of each test set of the file. This generated script includes (if needed) commands to regenerate the input, change the program file, recompile and reassemble it, execute the emulator, and extract the time from the output. When fed to sh, the commands produce a file, where each test set is appended with the time required to execute it. If we wanted several distinct times of the algorithm, e.g.,

62. Alternatively, a set of execution times, if we want to analyse different stages of an algorithm separately.

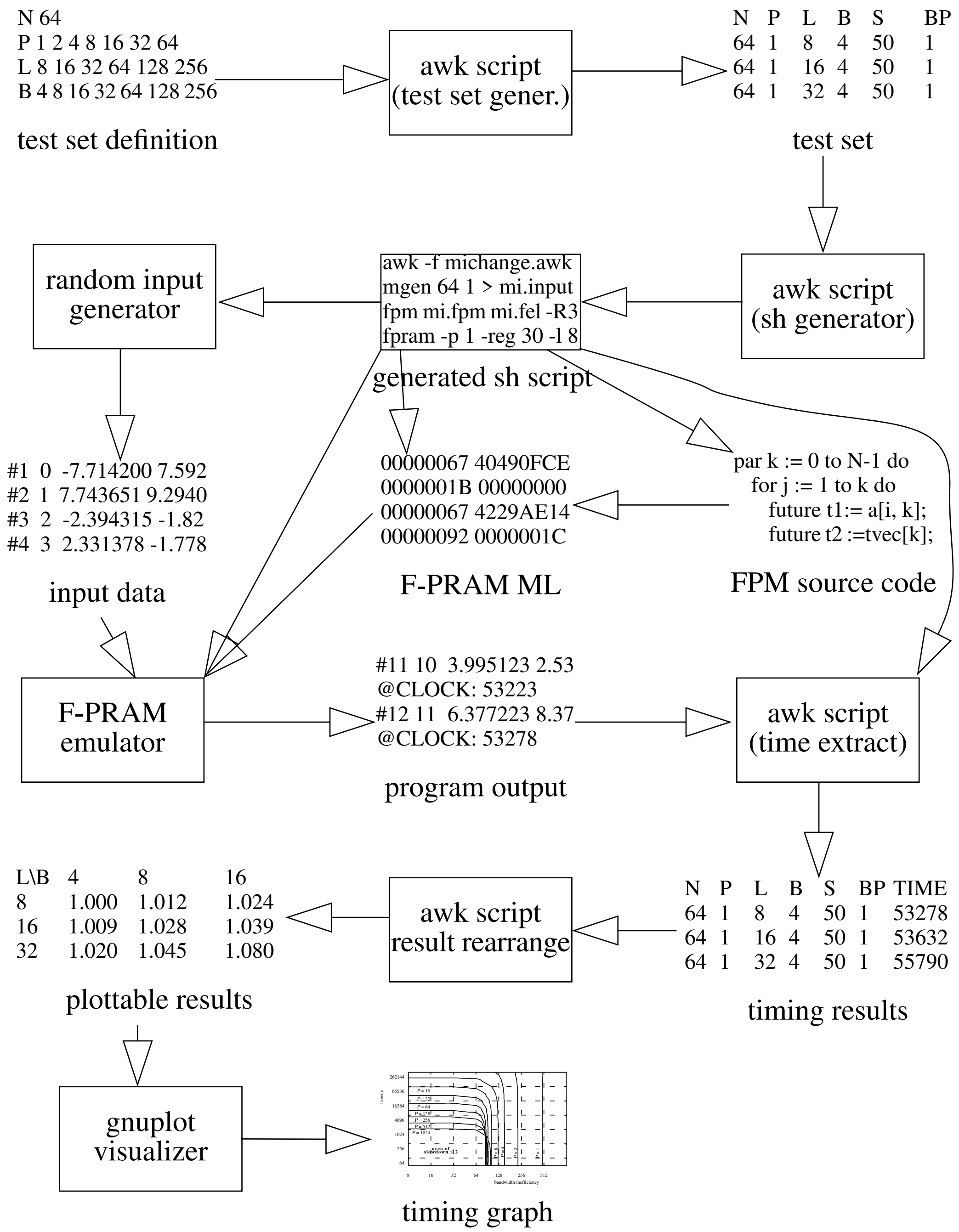


**Figure 6-6:** Automated measurement system for the F-PRAM emulator.

separate stages, we can append several times. We call the combination of the test set and corresponding time a *result set*. These result sets are often useful as they are, but usually we transform a file of result sets to a matrix of two parameters, which can be more easily plotted using a proper visualizer. Figure 6-6 presents these stages as a flow of transformations and operations from a test set definition to a drawn graph.

# Chapter 7

# Example algorithm implementations

A new theoretical model of any kind is of little value unless it is followed by a set of examples that demonstrate the usability of the new model. In this chapter we shall present a set of sample algorithms for the F-PRAM model. We shall present sorting, matrix multiplication, and matrix inversion programs, in Sections 7.1, 7.2, and 7.3, respectively. For each of the algorithms, we present the implementation on the F-PRAM model in form of the FPM language, analysis with the F-PRAM parameters, and measured results of the emulated programs. The measured results present the actual impact of the different F-PRAM parameters. Also, we compare the analysis and the measured results to improve our analysis methods. The matrix inversion is presented as a straightforward parallelization of an existing sequential program. Additionally, we shall present finding the maximum, software synchronization, and image smoothing in Sections 7.4, 7.5, and 7.6, respectively. We shall not give these simpler problems the same full treatment as for the first three ones. Instead we shall only point out the most interesting properties of these algorithms.

The main contribution of this chapter is the information on the critical parameters for each algorithm. For example, all algorithms tolerate bandwidth inefficiency or long latency up to some level. Beyond this threshold the algorithms will usually encounter severe performance penalty. The selected algorithms encounter quite different thresholds of the parameters and penalties caused by the features. We try to analyse this threshold behavior using applied analysis techniques, but the most evident results are achieved using measured performance on the F-PRAM emulator.

### Algorithmic notation used in the examples

The algorithms presented in this chapter have been implemented and executed as whole working programs within the F-PRAM emulator system. Within this chapter we present the algorithms first informally with a couple of words, and then as an accurately implemented algorithm. The actual source code of the programs would not, however, be the most readable version of the algorithms. Instead, we present here pretty-printed and compacted versions of the original FPM programs. By compacting we mean omitting some less important components of the program code. Especially we omit the variable declarations, “end” parentheses, some of the less important program fractions, such as most input and output routines, and temporary register variables used for intermediate results. Furthermore, we compact the most obvious iterations to vector operations. For example, an iteration

**for** i := 0 **to** N–1 **do** (7-1)
  **fwrite** y[i, j] := col[i];
**end**;

will be compacted to

**fwrite** y[0..N–1, j] := col[0..N–1]; (7-2)

which has to be understood as an iteration from 0 to $N$–1. In the later implementations the iteration should be replaced with a block reference when they are available. The emulated results use only iterations since the current emulator system does not support block references. To fit the algorithms in fewer lines, we also place two simple statements on a line, when possible. The resulting notation is a quite good compromise between the accuracy and readability. It corresponds closely with most notations found in algorithm text books. The notation is, however, directly obtained using a 1:1 mapping of the compileable source code. When introducing some of the algorithms, we first present a more abstract algorithm notation version to illustrate the algorithm.

### Notes on the analyses

In the analysis of the following algorithms, the asynchronous accesses of the shared memory yield many time complexity terms of the form $\max(C \times N, B \times N, L)$. This stands for the simultaneity of the computation (of length $C \times N$) and communication. The use of the constants is not a common practice in algorithm analysis. We use it to show, e.g., that the algorithm tolerates some latency without performance penalty. We shall not give any analysed values for the constants. Instead, we shall discover the same effects using the emulator system.

While analysing some F-PRAM algorithms, we shall use logarithms of nonconstant base, and transform them to quotients of logarithms of constant base. The transforms result in situations where the simplified formulas would give non-real values for real and reasonable boundary values of the parameters. Thus, in those situations we shall give some additional explanations on the usable domains of the formulas. Similar anomalies occur with transformations of logarithms of quotients and products to differences and sums of logarithms.

### Notes on the experiments and the result graphs

Most of the curves of the graphs in this chapter represent the impact of one (or two in some cases) parameter(s) on the execution time. All of the graphs include several curves for different values of another variable. Each of the curves is plotted separately based on some base value. For parameter $P$, the base value is $P = 1$, for other parameters the base value is small enough that it does not have any effect on the execution. The curves are plotted as either speedup or slowdown curves with respect to the first test run with the base value.

The speedups of $P > 1$ are calculated as ratios of the execution time of the case $P = 1$ to the case of $P > 1$. The slowdowns are calculated as ratios of the execution time of the current case to the case of the base value of the parameter.

Most of the measured graphs of the following algorithms feature a very large range of parameters. For example, the impact of latency often ranges from 16 to 256,000. To cover this large dynamics, we have to use logarithmic scales. The logarithmic scales apparently reduce some phenomena and might make some results look better than they really are. Therefore, we should carefully observe the scaling when examining the graphs. We can justify the logarithmic scales by the fact that in the computer industry, the most usual improvements are "doubling" or "halving" something, which results in exponential or logarithmic development, seen, e.g., in the Moore's law on microprocessors.

To draw the graphs with logarithmic $x$-axis, the experiments were run with exponential steps. In most cases the exponential step was 2. To draw the graphs with linear $x$-axis, the experiments were generally run with constant step 1, i.e., once for every value in the range. The few exceptions are mentioned in the text. In some cases we have omitted some (e.g., every other) of the measured curves to improve the clarity of the graphs.

All graphs in this chapter are drawn based on one test run for each combination of the parameters. The reasons for not drawing the graphs based on averages of several test runs were the long time needed for the experiments and the fact that none of the final programs showed significant alternation in the execution times. Repeating every experiment a significant number of times would have taken years of computer time. Instead, we tested the stability of each program separately for a smaller set of variables. As matrix multiplication, finding the maximum, and image smoothing algorithms are deterministic, all the variation is due to the alternating access times of the shared memory. In practice, in an unsaturated situation a processor will encounter (near) maximum latency. Consequently, the algorithms consume nearly the worst time on every run. In a case of saturated (full) interconnection network, the emulator delivers packets at full, thus deterministic, rate. The execution times of the mergesort and matrix inversion depend slightly on the input, but the variation due to the randomly generated input data was negligible. All the discovered variation of the execution times of all these algorithms remained below 1 %. The software synchronization routine was the only one which showed significant variation, as seen in Figure 7-26. The variation was, however, due to the randomly set presynchronization delays used in the experiment. The processors entered the procedure randomly (and, thus, the clocks were started randomly), but the execution time of the algorithm is a multiple of the time used by the inner synchronization iteration.

If the tests would have produced variable results, the graphs would contain irregularities because the different points of the plots are taken from different test runs. The graphs do not, however, have any significant irregularities that could not be explained separately. Consequently, we claim that all of the graphs in this chapter show fair results of the emulations even if they are not averages of several tests. The random number generator used to generate inputs was taken from [86] and it is supposed to produce good random numbers [64].

## 7.1 Odd-even mergesort

Sorting is probably the most used example problem in algorithmics. Also, in parallel computing the odd-even mergesort by Batcher [13] is one of the oldest parallel algorithms. Further, the pursuit of a work efficient optimal time algorithms was, and still is, quite a challenge [5]. In practical parallel computation the sorting problem is probably not one of the most important problems, but sorting is used, e.g., in database operations and decision-support systems. Also, the sorting makes an interesting example, which is why we shall present it here.

The number of existing parallel sorting algorithms is rather large. We can divide the algorithms in two classes, the classic comparison-based deterministic algorithms, and the other counting and/or sampling based algorithms, which often are more or less randomized. For sorting very large data sets, as is relevant with parallel computers, different sampling sort variants are currently quite popular, probably because they perform well when $N \gg P$. Because of the emulator system, we could not, however, measure sorting of billions of elements on hundreds or thousands of processors as, e.g., in [27]. Therefore, we had to settle for the more traditional algorithms that work well even with smaller inputs.

Our example belongs to the class of classic comparison-based deterministic algorithms since they parallelize well even with a smaller set of data. The optimal algorithms, such as the AKS [4] sorting network and Cole's parallel mergesort [22] sort $N$ elements in $O(\log N)$ time using $O(N)$ processors, i.e., they are asymptotically optimal (ENC class) for comparison based algorithms. We did not, however, choose any of these optimal algorithms because they are not optimal with the existing sets of data, as was noted by Natvig [80]. Instead, we chose the classic $O(\log^2 N)$ time odd-even parallel mergesort by Batcher [13]. Because of the straightforwardness of the algorithm, the constants of the execution are much lower than those of the asymptotically faster algorithms. Consequently, unless the set of data is unimaginably large, the factor $O(\log P)$ impacts less than the huge constants.

### The odd-even mergesort algorithm

The idea of mergesort is obvious, divide the input in two subsequences, recursively sort both of the subsequences, and merge the two sorted subsequences to achieve the whole sorted sequence. The recursion continues until there is only one element in the input sequence. Because one element is readily sorted, we can terminate the recursion. The number of levels of the recursion is $\log_2 N$. The running time of the algorithm is

$$T(N) = 2 \times T(N/2) + T_{merge}(N), \tag{7-3}$$

where $T_{merge}$ is the running time of merging of the two $N/2$ subsequences. Sequentially the merging can be trivially done in linear time. Thus, on each level of the recursion there is $O(N)$ work, or more formally

$$\begin{aligned} &T(N) = 2 \times T(N/2) + \mathrm{O}(N) \qquad \Rightarrow \\ &T(N) = O(N \log N), \end{aligned} \tag{7-4}$$

which is optimal for a comparison based algorithm.

```
procedure Odd-even_mergesort (A : array[1..N]);
    if Processors = 1 then
        Sequential_mergesort(A);
    else
        par i = 1 to 2 do
            Odd-even_mergesort(ith half of A);
        Odd-even_merge(halves of A);
    synchronize;

procedure Odd-even_merge (A : array[1..N]);
    if Processors = 1 then
        Sequential_merge(A);
    else
        par i = 0 to 1 do
            Odd-even_merge(halves of odd/even (2n+i) elements of A);
        par i = 2 to N by 2 do
            pipelined_compare-exchange (A[i], A[i+1]);
    synchronize;
```

**Algorithm 7-1:** Odd-even mergesort.

The parallel versions of the mergesort trivially parallelize the recursive sorting of the two subsequences. In this way we can easily exploit $O(N)$ processors, but do not gain much speedup since the sequential merging takes linear time.[63] Consequently, we have to parallelize also the merging phase, which is the point where the different parallel mergesort algorithms diverge.

In the odd-even merge, the merging is also done recursively. More specifically, given two sorted sequences $A$ and $B$, we firstly combine them to a sequence $AB$. Secondly, we recursively merge the halves of odd elements of the $AB$ and the halves of the even elements of the $AB$. After that, each element is by at most one place out of its correct position. Thus, thirdly, we can compare all $(2i–1, 2i)$ pairs of the elements in parallel to complete the merging. All these stages fit into a butterfly network, i.e., all data dividing and merging is done normally[64] along the connections of a binary butterfly.

In all stages we proceed sequentially as soon as we run out of processors. Algorithm 7-1 presents the sorting pseudocode. The running time of the merging stage is

$$T_{oemerge}(N, P) = \begin{cases} T_{oemerge}(N/2, P/2) + N/P & \text{if } P > 1 \\ N & \text{if } P = 1 \end{cases} \tag{7-5}$$

which can be solved to the form

63. Thus, in this version the whole sorting takes also linear time.

64. Using only one level of nodes and connections at a time.

$$T_{oemerge}(N,\ P) = (\log P+1)\times N/P = O(N\log P/P)\ . \tag{7-6}$$

The complexity of the sorting is the same as before,

$$T(N,\ P)\ =\ \begin{cases} T(N/2,\ P/2) + T_{merge}(N,\ P) & \text{if } P > 1 \\ N\log N \quad \text{if } \ P = 1 & , \end{cases} \tag{7-7}$$

which can be solved to the form

$$T(N,\ P) = \log P\times(\log P+1)\times N/P + (N/P)\times\log(N/P) \tag{7-8}$$

when assigned the complexity of the odd-even mergesort. The asymptotic complexity is thus

$$T(N,\ P) = O((N/P)\times(\log^2 P + \log N/P)), \tag{7-9}$$

which equals the familiar

$$T(N) = O(\log^2 N) \quad \text{if } P/2 \geq N, \tag{7-10}$$

i.e., we exploit the full parallelism of the algorithm. Compared with the optimal solution, Formula (7-9) has one additional $\log^2 P$ term slowing down the speedup as the number of processors increases. The term originates from the fact that the odd-even communication within the processors goes back-and-forth along the butterfly during the recursive steps. To keep the inefficiency moderate, e.g., at 50 %, we can balance the additive terms of Formula (7-9) using inequality

$$\begin{aligned} &\log^2 P \leq \log N/P && \Leftrightarrow && (7\text{-}11) \\ &\log N \geq \log P + \log^2 P && \Leftrightarrow \\ &N \geq 2^{\log P + \log^2 P} && \Leftrightarrow \\ &N \geq P \times P^{\log P} && \Leftrightarrow \\ &N \geq P^{\log P + 1}\ , && && (7\text{-}12) \end{aligned}$$

which is quite a strong requirement. For example, if $P = 64$, then $N$ should be at least $4.4\times10^{12}$, more than the number of all people who ever lived on the earth! In practice, we can use more processors, since the constants of the different parts of the algorithm differ, and the small variations on the constants impact exponentially in formulas such as (7-14). As well as we solved $N$ from Inequality (7-11), we can solve $P$

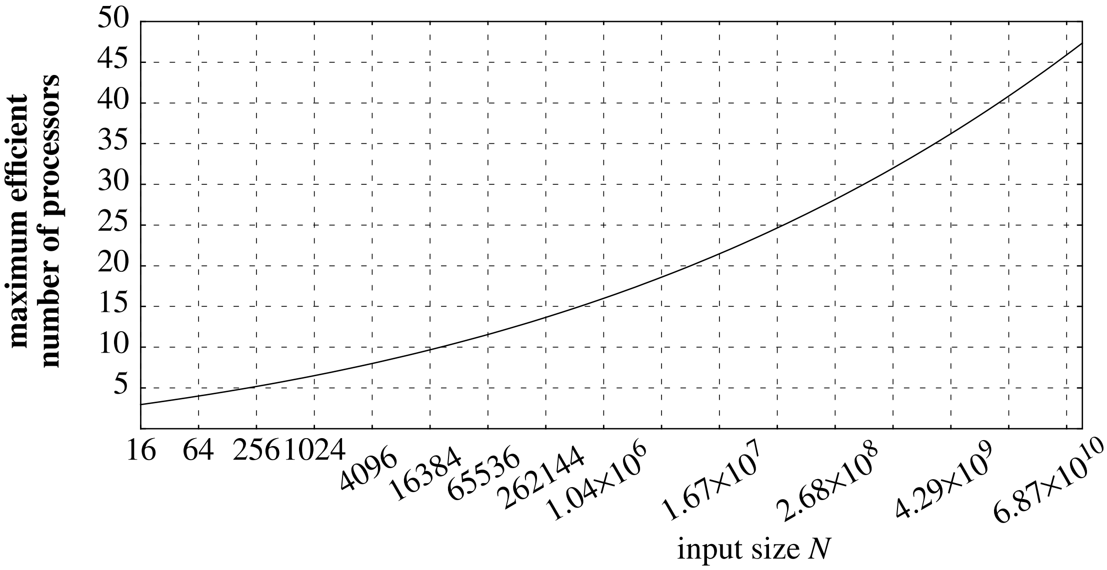


**Figure 7-1:** Maximum efficiently useful $P$ as a function of $N$ as predicted by Formula (7-14), odd-even mergesort, logarithmic x-axis.

$$\log^2 P \le \log N/P \quad \Leftrightarrow \tag{7-13}$$

$$\log P + \log^2 P - \log N \le 0 \quad \Leftrightarrow \quad | \; x := \log P$$

$$x^2 + x - \log N \le 0 \quad \Leftrightarrow$$

$$x \le \frac{-1 \pm \sqrt{1 + 4\log N}}{2} \quad \Leftrightarrow$$

$$x \le \frac{-1 + \sqrt{1 + 4\log N}}{2},$$

since $x > 0$. By assigning $x$ back we get

$$\log P \le \frac{-1 + \sqrt{1 + 4\log N}}{2} \quad \Leftrightarrow$$

$$P \le 2^{\left(\frac{-1 + \sqrt{1 + 4\log N}}{2}\right)} \tag{7-14}$$

$$\le 2^{\sqrt{\log N}}, \tag{7-15}$$

which is, however, a bit too complicated to be used in a program to decide a good number of processors to be used. Figure 7-1 presents the right side of Formula (7-14) plotted in the practical application range up to $N = 2^{36}$.

**F-PRAM implementation**

The key points of the F-PRAM implementation are correct fetching of the elements of the shared input array to the local memories, and the sharing of the intermediate results via

```
procedure msort (sharedvar S : array of word; First, Length : word);
    if (Length > 2) and (P > 1) then
        par i := 0 to 1 do
            msort(S, First + i * Length/2, Length/2 );
        oemerge(S, First, Length, 1);
    else
        seqmsort(S, First, Length);
    synchronize;
```

**Algorithm 7-2:** Parallel mergesort procedure.

the shared memory. Here we shall present the parallel mergesort and merge procedures, but omit the sequential versions, as well as I/O. Since the procedures use index arithmetics following the butterfly network, we do not use the vector reference notation, but present all iterations explicitly.

The main mergesort procedure, shown as Algorithm 7-2, is very short since it does not need to care about, e.g., interprocessor communication. The procedure only checks whether to use parallel or sequential mergesort. The rule is based on the length of the vector and the number of available processors. For a very refined solution, we could have more choices, e.g., stop the recursion at $Length \leq 4$, and use brute force *if-elsif-else* sort for the short sequence. In practice, these tricks would not, however, improve performance significantly.

The merging procedure, shown as Algorithm 7-3, is slightly more complicated than main sorting procedure since it involves actual comparisons between elements. As in the main sorting procedure, the merging procedure first checks the trivial cases (lines 2-8). The main parallel merging starts with the parallel recursive call to merge the halves of odd and even components of the vector (lines 10-11). The rest of the procedure implements the parallel compare-exchange operation. The straightforward fetching of a pair of values, comparing them, and possibly rewriting them (lines 24-30), works well unless the latency is high. If the latency is high and each processor needs to do more than one compare-exchange operation, we can do better. We first fetch all the values the processor needs (lines 16-18), and after that do the comparisons locally, possibly rewriting the changed pairs (lines 19-22). This way we can change the multiplicative factor $L$ to an additive factor $L$, which is beneficial even with more complex logic if the latency is high. The boundary $L = 50$ was chosen experimentally.

### F-PRAM analysis with $B$ and $L$

The complexity of Formula (7-9) holds if we do not take shared memory access costs into account. Here we repeat the analysis with the most important parameters $B$ and $L$. The running time formula of the parallel mergesort main procedure

```
procedure oemerge(sharedvar S : array of word; First, Length, Interl : word);
    if P < 2 then
        seqmerge(S, First, Length, Interl);
    else
        if Length <= 2 then
            future a := S[First];   future b := S[First + Interl];
            if a > b then
                fwrite S[First] := b;   fwrite S[First + Interl] := a;
        else
            par i := 0 to 1 do
                oemerge(S, First + i * Interl, Length/2, Interl * 2);
            if (P < Length/2) and (L > 50) then
                Length2 := Length / P;
                par i := 0 to P-1 do
                    j := i * Length2 + 1;
                    for k := 0 to Length2 - 1 do
                        if (j + k + 1) < Length then
                            future LocS[k] := S[First + ((j + k) * Interl)];
                    for k := 0 to Length2 - 1 by 2 do
                        if ((LocS[k]) > LocS[k+1]) and (j+k+1 < Length) then
                            fwrite S[First + ((j+k) * Interl)] := LocS[k+1];
                            fwrite S[First + ((j+k+1) * Interl)] := LocS[k];
            else
                par i := 1 to Length/2 - 1  do
                    j := i * 2;
                    future a := S[First + (j - 1) * Interl];
                    future b := S[First + j * Interl];
                    if a > b then
                        fwrite S[First + (j - 1) * Interl] := b;
                        fwrite S[First + j * Interl] := a;
    synchronize;
```

**Algorithm 7-3:** Parallel odd-even merge procedure.

$$T(N, P) = \begin{cases} T(N/2, P/2) + T_{merge}(N, P) & \text{if } P > 1 \\ T_{seqsort}(N) & \text{if } P = 1 \end{cases} \tag{7-16}$$

does not change since the procedure does not use shared memory.[65] The running times of the subroutines, however, have to be reanalysed. The basic $O(N\log N)$ time of the sequential mergesort is increased by the cost of fetching the array to the local memory, and storing it back to the shared memory. Consequently,

65. It only delivers pointers to the shared memory for the subroutines.

$$T_{seqsort}(N) = N\log N + 2\times\max(N\times B,\ L). \tag{7-17}$$

Similarly, the complexity of the sequential merging is now

$$\begin{aligned} T_{seqmerge}(N) &= C_{sm}\times N + 2\times\max(N\times B,\ L) \\ &= O(N\times B + L), \end{aligned} \tag{7-18}$$

where $C_{sm}$ is a local constant time needed for a merge step. Note, that the proportional weight of the shared memory access is greater here than in sequential sorting.

The complexity of the parallel merge is slightly more complicated. The recursion formula is

$$\begin{aligned} T_{oemerge}(N,\ P) = {} & T_{oemerge}(N/2,\ P/2) + \max(B\times N/P,\ L) + \\ & \qquad \max(C_{oem}\times N/P,\ L) \\ \le {} & \begin{cases} T_{oemerge}(N/2,\ P/2) + 2\times\max(B\times N/P,\ C_{oem}\times N/P,\ L) & \text{if } P > 1 \\ T_{seqmerge}(N) & \text{if } P = 1. \end{cases} \end{aligned} \tag{7-19}$$

By solving the recursion we get

$$\begin{aligned} T_{oemerge}(N,\ P) = {} & 2\times\log P\times\max(B\times N/P,\ C_{oem}\times N/P,\ L) + \\ & \qquad C_{sm}\times N/P + 2\times\max((N/P)\times B,\ L) \end{aligned} \tag{7-20}$$

$$\begin{aligned} & \le 2\times(\log P{+}1)\times\max(B\times N/P,\ C_{oem}\times N/P,\ L) \\ & = O(\log P\times(B\times N/P + L)). \end{aligned} \tag{7-21}$$

Applying the new merging complexity to the sorting complexity, we get

$$T(N,\ P) = \begin{cases} T(N/2,\ P/2) + (\log P{+}1)\times\max(B\times N/P,\ C_{oem}\times N/P, L) & \text{if } P > 1, \\ N\log N + 2\times\max(N\times B,\ L) & \text{if } P = 1 \end{cases} \tag{7-22}$$

$$\begin{aligned} & = (\log P)(\log P{+}1)\times\max(B\times N/P,\ C_{oem}\times N/P,\ L) + (N/P)\log(N/P) + \\ & \qquad\qquad 2\times\max((N/P)\times B,\ L) \\ & \le (\log P{+}1)^2\times\max(B\times N/P,\ C_{oem}\times N/P,\ L) + (N/P)\log(N/P) \end{aligned} \tag{7-23}$$

$$= O(\log^2 P\,(B\times N/P + L) + (N/P)\log(N/P)). \tag{7-24}$$

Now we can see the expected fact that the shared memory access costs affect multiplicatively on the costs induced by the parallel sorting, but only additively on the cost of the sequential sorting. Studying Formula (7-24), we could conclude that any latency or bandwidth inefficiency would reduce the efficiency of the algorithm radically. This is not true, as we can see from Formula (7-23). Up to some limit, the algorithm tolerates both latency and bandwidth inefficiency while doing other operations. In other words, the $O$-notation hides the constants of the previous formulas. These constants show us some important properties of the algorithm, especially the tolerance of some latency and bandwidth inefficiency. The measured results later in this section confirm these conclusions.

### Measured performance on the emulator system

Since the algorithm switches to sequential mergesort as soon as it runs out of processors, the speedup comparisons are made against the sequential mergesort. As opposed to quick-

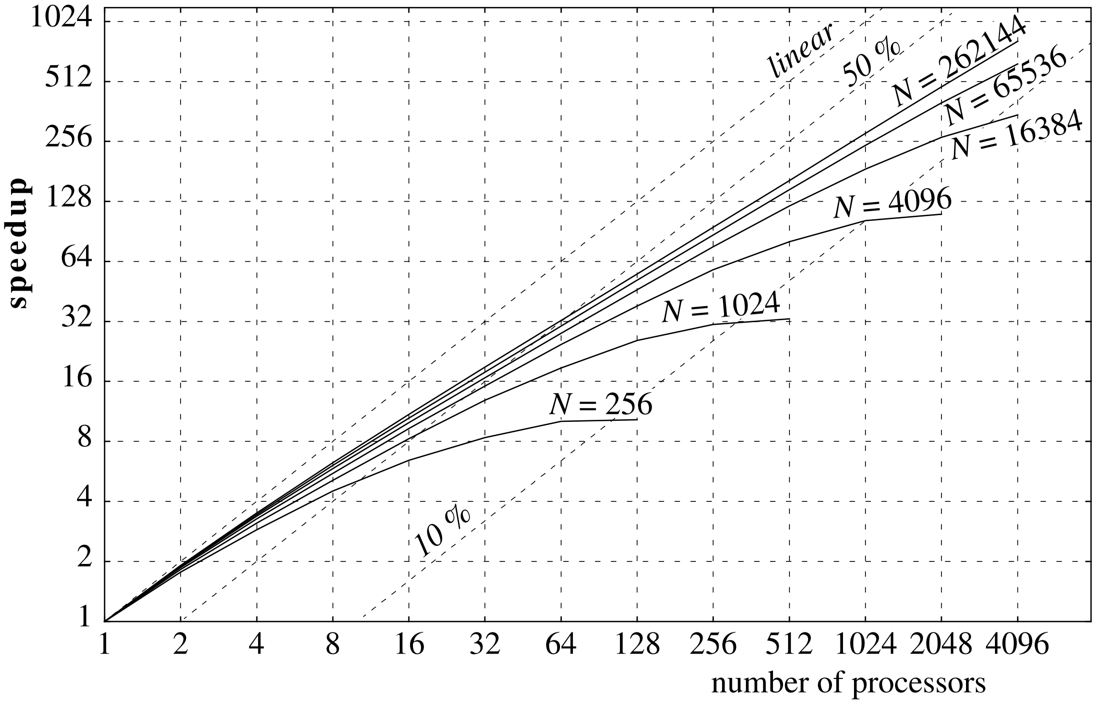


**Figure 7-2:** Speedup of odd-even mergesort as a function of the number of processors for different input sizes. Both scales are logarithmic.

sort, mergesort is a better counterpart in comparisons since it deterministically executes in $O(N \log N)$ time.

In the full working program, the input is read to the shared memory, and sorted in place.[66] During the sequential stages, the data is read to the local memory, sorted/merged, and written back to the shared memory. The measured times include only the sorting time, not the trivially parallelizable input and output. The inputs for the program were sets of random numbers. Sorting strings would make the compare-exchange phase slower, and, thus, reducing the proportional impact of communication and thereby improving the overall parallelizability of the algorithm.

To begin with, we shall present the basic performance of the implementation in an optimal ($B = L = 1$) parallel computer. Figure 7-2 presents the speedup as a function of the number of processors. Different lines present the speedup curves measured with different input sizes. The speedup is measured relative to the execution of the same algorithm using only one processor. To ease the readability, the figure also includes lines for linear speedup, i.e., 100 % efficiency in parallelization, 50 % efficiency, and 10 % efficiency. As an example we can conclude that when sorting 264144 ($2^{18}$) elements, we have efficiency at least 50 % if $P \leq 64$. For $P = 4096$, the speedup is about 815, which stands for approximately 20 % efficiency. These numbers differ a bit from the efficiency requirement given in Formula (7-12) because these are measured with the actual constants involved.

The speedup curves here are compared with the $P = 1$ execution of the same program. This version inputs the data to the shared memory, sorts it by copying it back to the

66. Without additional shared arrays.

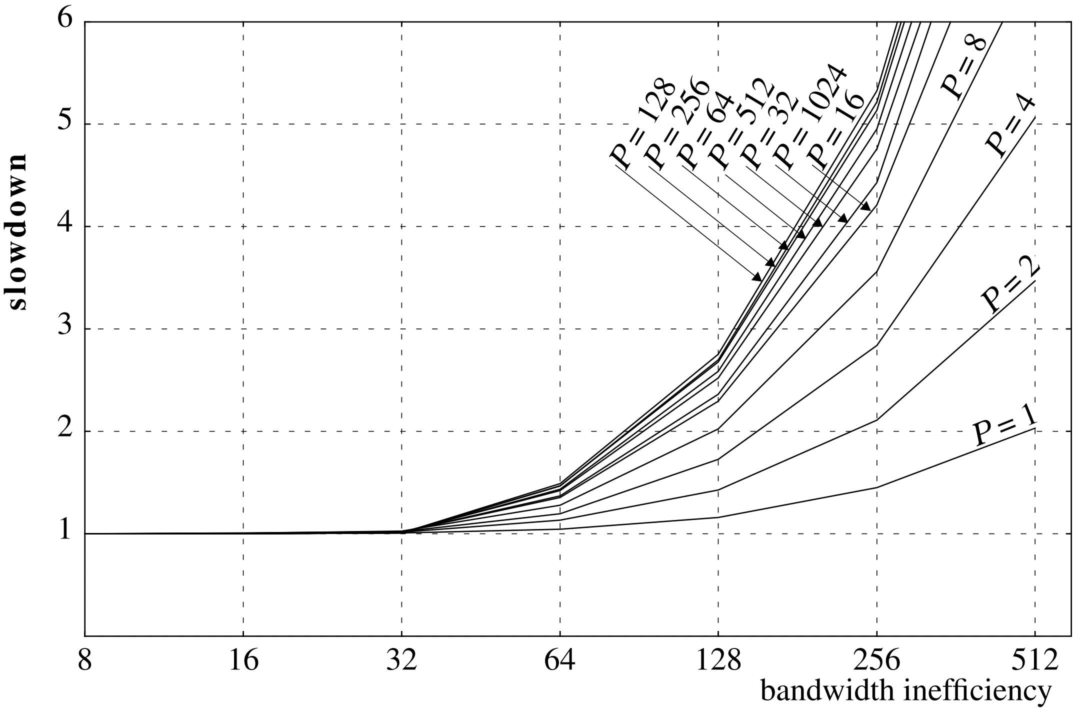


**Figure 7-3:** Slowdown of the odd-even mergesort when the bandwidth inefficiency increases. The numbers of processors varies, $N = 16384$.

local memory, and writes it back to the shared memory. This can be viewed either as an unfair trick to improve the speedups, or as a reasonable assumption for the sake of consistency. The latter explanation is valid if we use the sorting as a subroutine of a larger program, and try to decide whether to parallelize or not. The actual impact of the shared memory access is not very large when we use only one processor. For example, when sorting 16384 elements with one processor, the whole algorithm using the shared memory takes $1.443{\times}10^7$ clock cycles. By excluding the shared memory access, i.e., only sorting a local array, it takes $1.363{\times}10^7$ clock cycles. Consequently, we could adjust the speedup numbers by 5.9 % if we would consider the comparisons unfair. For bigger sets of data the relative difference is even smaller. Similarly, the impacts of $B$ and $L$ in case of $P = 1$ should be ignored if the sorting program would be used as stand-alone program.

### The impact of latency and bandwidth inefficiency

Considering the impact of the shared memory access parameters, we ran the same 16384 (16k) element sorting task varying the parameters $P$, $B$, and $L$ forming a 3-dimensional domain of executions. The following graphs present the results of these experiments. Figure 7-3 presents the impact of bandwidth inefficiency $B$ for different values of $P$ with low $L$. The bandwidth inefficiency starts to impact beyond $B = 32$, and impacts significantly beyond 64. The volume of shared memory accesses increases as the number of recursive parallel steps increases. On the other hand, also the total bandwidth of the computer increases, which balances the impact of increased communication. Thus, the uppermost lines representing $P = 16..1024$ behave quite similarly. As $B$ and $P$ are of the same order, the modulo/rounding variation even changes the order of the lines.

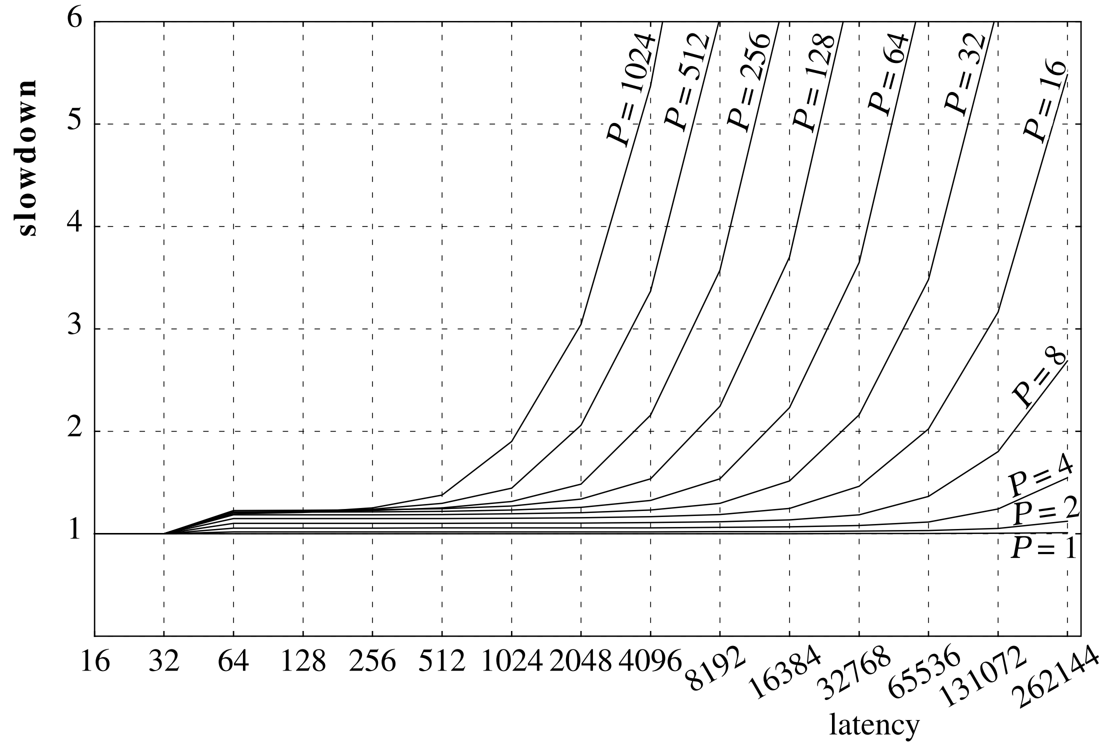


**Figure 7-4:** Slowdown of the odd-even mergesort when the latency increases. The numbers of processors varies, $N = 16384$.

Figure 7-4 presents the impact of latency $L$ for different values of $P$ with low $B$. For low values of $P$ the latency does not impact performance significantly, as expected. Because for larger values of $P$ the number of recursive parallel steps increases, the longest path of communications increases, and, thus, the latency tolerance decreases significantly. The step between $L = 32$ and $L = 64$ for larger values of $P$ is due to the more efficient but non-latency-tolerating compare-exchange used if $L \leq 50$. The sharpness of the step is due to the measurements only at points $L = 32$ and $L = 64$.

Figure 7-5 presents a conclusive version of the combined impact of $B$ and $L$ for the same sorting using different numbers of processors. The graph presents a 2-dimensional space of parameters $B$ and $L$. For each $P$ there is a line to present the boundary of efficient execution compared to the optimal execution with the corresponding $P$. The sets of $B$ and $L$ that are below and left of the line enable the algorithm to execute in at most twice the time of the optimal execution for that $P$. The pairs of $B$ and $L$ that are above and right of the line induce more than a 2-fold slowdown. The lines are drawn as slowdown = 2 contour lines from 3-dimensional surfaces plotted from the data for each $P$.

The graph of Figure 7-5 does not take into account the slowdown induced by the parallelization inefficiency. If we combine the results of Figures 7-2 and 7-5, we get the requirements for the total 50 % efficiency. In other words, instead of comparing each execution time to the execution time of an optimal machine with the same $P$, we compare each work to the work of an optimal machine with $P = 1$. Figure 7-6 presents this version with the same data as Figure 7-5. Now all the pairs of parameters $B$ and $L$ which are left/below a line enable the algorithm possible to execute with at most twice the work than with a machine with $P = B = L = 1$. Note that using $P > 32$, $N = 16k$, our implementation did not

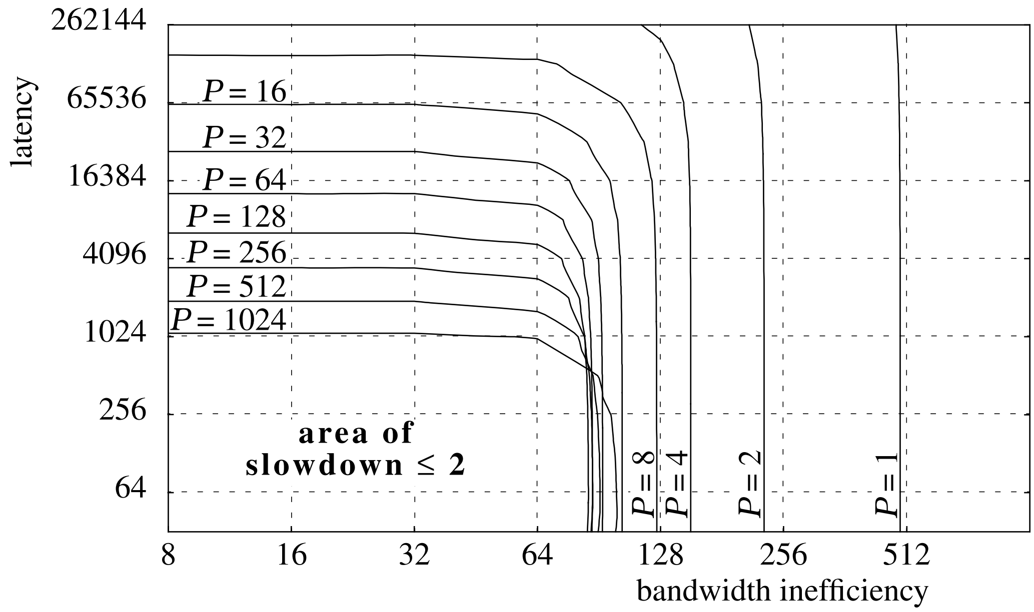


**Figure 7-5:** Acceptable (at most twice the time of the execution in an optimal machine with the same $P$) latencies and bandwidth inefficiencies in odd-even mergesort. The numbers of processors varies, $N = 16384$.

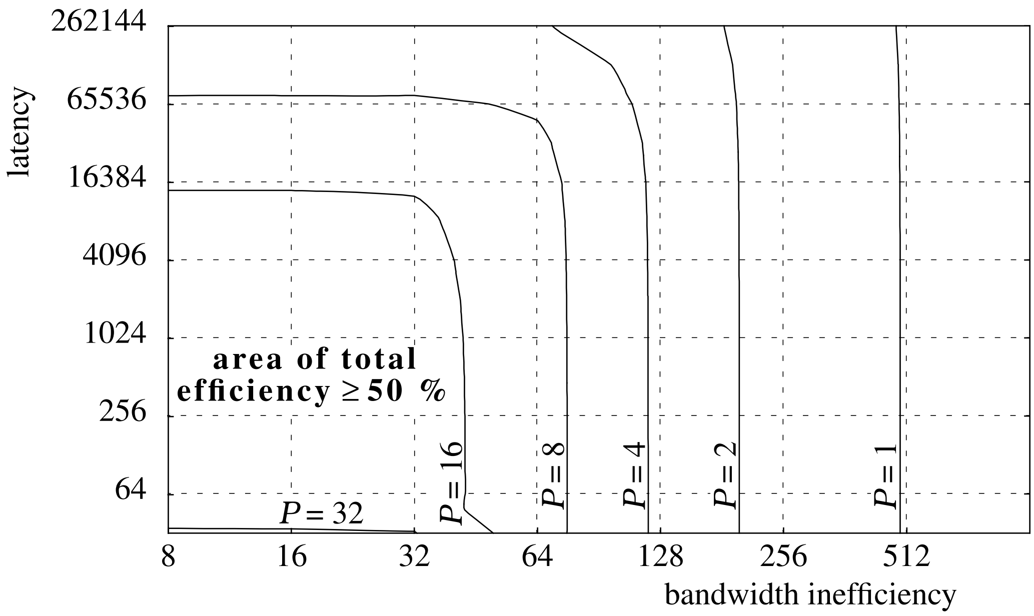


**Figure 7-6:** Requirements on bandwidth inefficiency, latency, and number of processors for efficient (at most twice the work of the sequential execution in an optimal machine) execution of odd-even mergesort, $N = 16384$.

meet the 50 % efficiency requirement at all. The slight local irregularity in the graph is due to the different compare-exchange routine used if $L \leq 50$.

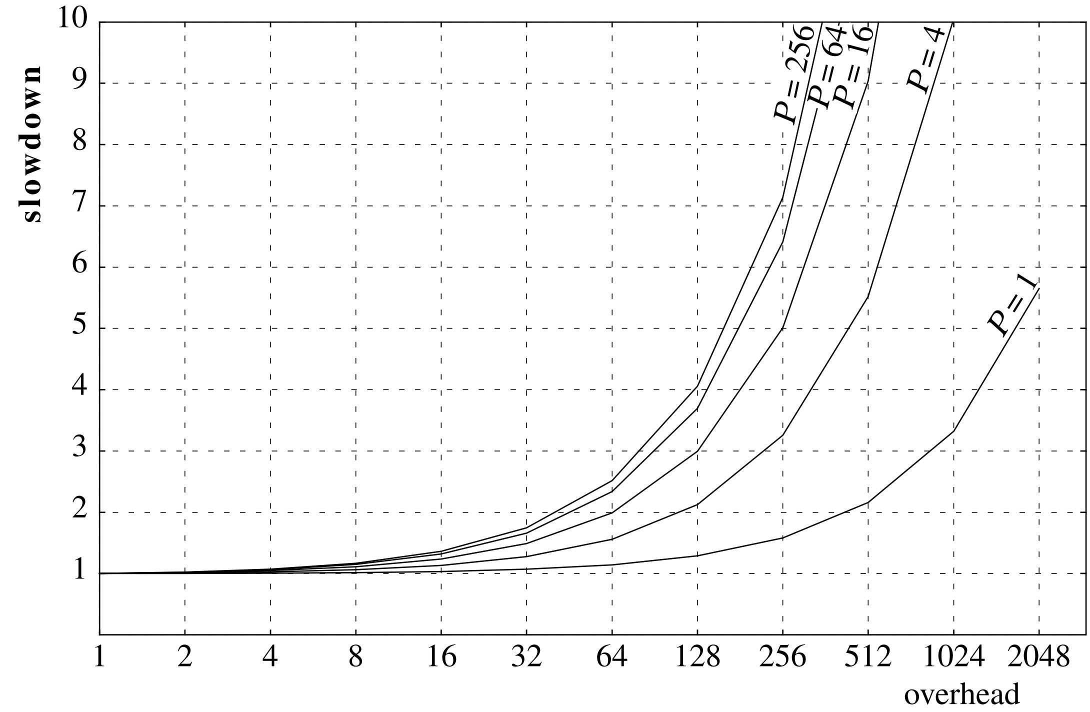


**Figure 7-7:** Slowdown of the odd-even mergesort when the overhead increases. The numbers of processors varies, $N = 16384$.

### The impact of overhead $B_P$

The previous analyses and measurement results assumed default values of the secondary parameters, such as the overhead $B_P$. When considering the impact of $B_P$, we have to count the number of shared memory references made by each processor. In practice, the processors communicate during the recursive stage of the merging and sorting. By examining Formula (7-23), we can directly insert the parameter $B_P$ into it. Thus, we get

$$T(N, P) = (\log P + 1)^2 \times \max(N \times B/P, (C_{oem} + B_P) \times N/P, L) + (N/P) \times \log(N/P), \quad (7\text{-}25)$$

which reveals that unless $B$ or $L$ is large, the $B_P$ will eventually impact the parallel term of the execution time multiplicatively.

Experimentally, we executed the odd-even mergesort with different overheads and different numbers of processors. The measured results can be seen in Figure 7-7. The graphs present the relative execution time, i.e., slowdown, compared to the $B_P = 1$ situation. For each different $P$ there is a curve as a function of $B_P$. We can see that even for moderate parallelism, such as $P = 16$, overhead $B_P$ as low as 64 is significant (100 % increase in execution time). Compared to the impact of latency seen in Figure 7-4, the impact of overhead is very significant even in small values. In other words, the approach of prefetching the shared memory references with the future primitive allows long latencies compared to immediate waiting for the referenced values. Here again, the curve for $P = 1$ can be considered futile.

When studying the values of F-PRAM parameters on current parallel computers, we shall find that the current programming libraries do not perfectly support the future, i.e., the asynchronous shared memory get. According to the above comparison on the impacts of latency and overhead, the parallel machine libraries should really support the asynchronous shared memory get.

## 7.2 Matrix multiplication

Multiplication of two matrices is defined as follows:

$$C = A \cdot B, \qquad c_{ij} = \sum_{k=0}^{N-1} a_{ik} \times b_{kj}, \tag{7-26}$$

where $N$ is the number of columns of the matrix $A$ and the number of rows of the matrix $B$.[67] For simplicity, we shall discuss here only square matrices, hence all of the matrices $A$, $B$, and $C$ are $N{\times}N$ matrices. Using the definition (7-26), we can trivially compute each element of the result matrix $C$ by $N$ multiplications and $N$–1 additions. Consequently, we need $O(N^3)$ multiplications and additions to compute the whole matrix $C$. Using more sophisticated techniques, we could do the multiplication with only $O(N^{\log_2 7})$ multiplications [96], or even faster. The asymptotically faster algorithms are not, however, practical for small to moderate sized matrices [24]. Moreover, the naive algorithm has an easily optimizable innermost iteration, whereas the more sophisticated ones are recursive and do not necessarily provide as long inner iterations. In other words, not only the algorithmic constants were larger, but also implementation constants would grow if we used the more optimal algorithms.

The computation of Formula (7-26) parallelizes trivially up to $N^2$ processors since each element of the result $C$ can be computed independently. Even the computation of each result element could be parallelized more by using up to $N$ processors to perform the multiplications in parallel, and adding each $c_{ij}$ using binary tree summing. Thus, the whole algorithm would need $N^3$ processors and execute in $O(\log N)$ time. This, however, would be inefficient by factor $O(\log N)$. More efficiently, we could do the task in $O(N/\log N)$ time using $N^2\log N$ processors. On the other hand, we rarely have more than $N^2$ processors to use, and the faster algorithm would again have larger constants. Moreover, the resulting algorithm with binary trees to compute the final sums would have rather severe $B$ and $L$ requirements for the interconnection network. Therefore, we restrict the parallelization to the trivial $N^2$ processors.

### F-PRAM implementation

Implementing Formula (7-26) in FPM is straightforward. The key points are distributing the computation of the result $C$ to the participating processors, and fetching the correct data for each processor. We chose the trivial (and optimal) distribution of assigning each processor a square of the result matrix to be computed. To compute a block of the result, each processor needs a set of rows from the matrix $A$, and a set of columns from the matrix $B$. Figure 7-8 presents the blocks of data needed by one processor to compute the black-

67. The common mathematical convention is to index the matrix rows and columns from 1 to $N$. Here, however, we use the 0..$N$-1 indexing to maintain coherence with the program code. The programs use the 0..$N$-1 for improved efficiency.

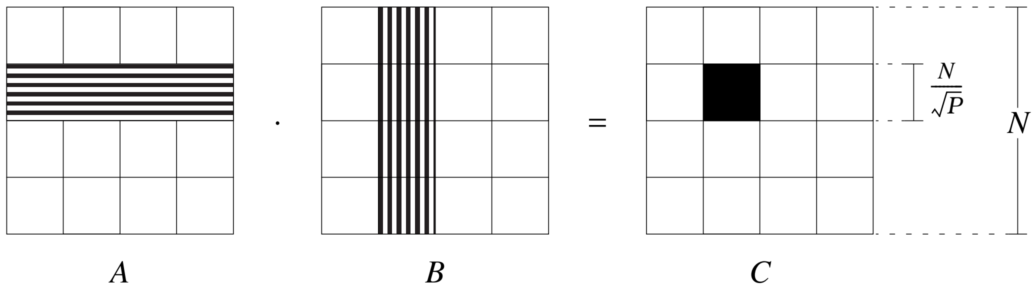


**Figure 7-8:** Data requirements of each processor in matrix multiplication.

```
1  procedure matmul (sharedvar A, B, C : matrix);
2      sproc = sqrt(P);   block = N / sproc;
3      par x := 0 to sproc - 1 do
4          par y := 0 to sproc - 1 do
5              for i := 0 to N-1 do          // prefetch a slice of A
6                  for j := 0 to block-1 do
7                      future locA[i, j] := A[i, y*block + j];
8              for i := 0 to block-1 do      // prefetch a slice of B
9                  for j := 0 to N-1 do
10                     future locB[j, i] := B[x*block + i, j];
11             for i := 0 to block-1 do
12                 for j := 0 to block-1 do
13                     ka := ?locA[0, j];      kb := ?locB[0, i];
14                     sum := 0;               kf := ?locA[N - 1, i];
15                     while (ka <= kf) do
16                         inc(sum, ka^ * kb^);
17                         inc(ka);      inc(kb);
18                     fwrite C[x*block + i, y*block + j] := sum;
```

**Algorithm 7-4:** Matrix multiplication in FPM.

ened block of the result matrix $C$. Each processor first issues the futures to fetch all the data it needs, and then does the computation locally. The writing to the result matrix is done as soon as a result is computed. Algorithm 7-4 presents the compacted source code of the multiplication procedure. The innermost iteration (dot product) of the multiplication is coded using pointers to arrays. The question marks (?) stand for “address of,” and the circumflexes (^) stand for “value in the address.” The ?-operator is a nonstandard but useful extension to the Modula-2 language. To make the innermost iteration as efficient as possible, the local copies (*LocB*) of the slices of the matrix $B$ are stored transposed. This way the successive elements of the columns are more efficiently accessed as they are located in successive memory locations. The compiler stores the matrices row by row, i.e., the elements $X[i, j]$ and $X[i{+}1, j]$ are stored in adjacent memory locations.

The local multiplication stage of the procedure (lines 11-17) takes

$$O\left(\left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 \times N\right) = O\left(\frac{N^3}{P}\right) \tag{7-27}$$

time, if $P \leq N^2$. During the prefetching stage (lines 5-10), each processor requests

$$2 \times \left\lceil \frac{N}{\sqrt{P}} \right\rceil \times N = O\left(\frac{N^2}{\sqrt{P}}\right) \tag{7-28}$$

words of data. The total execution time needed by the algorithm is thus

$$\max(2 \times \left\lceil \frac{N}{\sqrt{P}} \right\rceil \times N \times \max(C_{fe}, B), L) + \max(C_{mu} \times N, B) \times \frac{N^2}{P} + L, \tag{7-29}$$

where $C_{fe}$ is the local constant of the prefetching iterations, $C_{mu}$ is the constant of the dot product. The last $L$ and $B$ of the second last term are due to writing the result elements to the shared result matrix (line 18). By combining the maximum alternatives of Formula (7-29) and applying the $O$-notation, we get a rather obvious complexity

$$O\left(\frac{BN^2}{\sqrt{P}} + \frac{N^3}{P} + L\right). \tag{7-30}$$

Assuming $B = L = 1$, we can balance the two remaining terms to achieve a 50:50 time usage between the communication and the computation. To ensure 50 % asymptotic efficiency, it suffices that $P \leq N^2$, which holds always since our algorithm does not exploit any more parallelism. In our current implementation, however, the sum of the constants in the two prefetching phases of the algorithm is larger than the constant of the dot product. Consequently, using $N^2$ processors yields less than 50 % efficiency. We could improve the prefetching slightly by using similar pointer referencing as in the dot product, but significant improvements could be gained using block references. In a machine with small $B_B$, the improvement in the prefetch stage would be up to $26/B_B$ -fold.[68]

Studying Formula (7-29), the bandwidth inefficiency $B$ impacts the prefetching phase when it is larger than a small constant. The effect is multiplicative for the prefetching stage, and will impact the performance of the whole algorithm significantly if $P$ or $B$ is large. Because the latency $L$ impacts purely additively, it does not impact a lot, unless $P$ is close to $N^2$, i.e., the execution time is small, and the latency is very large.

### Measured performance on the emulator system

In measuring the performance of the matrix multiplication algorithm, we had the same assumptions and test setup as we had with the sorting algorithm presented in Section 7.1.

68. The body of the inner for-do loop of the prefetching is 26 clock cycles long using the current compiler and emulator system.

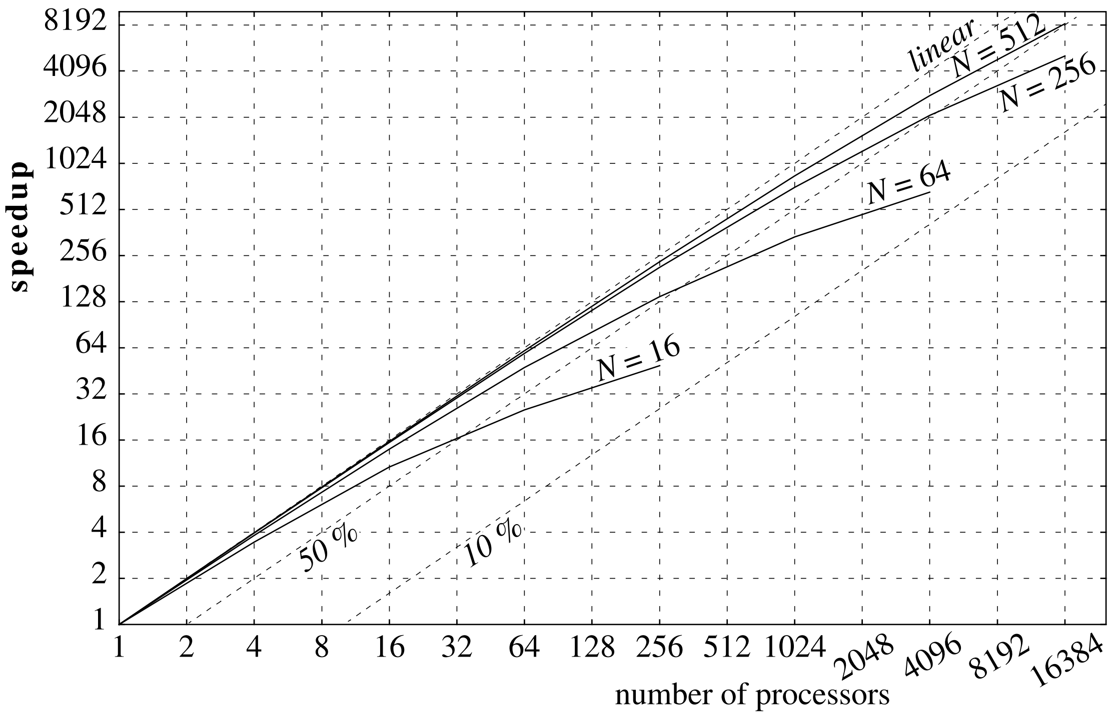


**Figure 7-9:** Matrix multiplication speedup as a function of the number of processors for different input sizes.

Figure 7-9 presents the speedup as a function of the number of processors. Different lines present the speedup curves measured with different input sizes. The input size $N$ stands for $N{\times}N$ matrices. The speedups are measured relative to the execution of the same algorithm using only one processor. Note that the scales are logarithmic. The figure also includes lines for 100 %, 50 %, and 10 % efficiency. We can conclude, e.g., that when multiplying 256×256 matrices, we have efficiency at least 50 %, as long as $P \leq 4096$, i.e., each processor has at least 16 elements of the result matrix to compute. For $P = 16384$ and $N = 512$, the speedup is about 8330, which equals to 51 % efficiency.

Considering the impact of the shared memory access parameters, we ran the same 64×64 matrix multiplication varying the parameters $P$, $B$, and $L$ forming a 3-dimensional domain of executions. The following graphs present the results of these experiments. Figure 7-10 presents the impact of bandwidth inefficiency $B$ for different values of $P$ with low $L$. The bandwidth inefficiency starts to impact when $B > 64$. The impact is significantly especially when $P$ is large. Figure 7-11 presents the impact of latency $L$ for different values of $P$ with low $B$. We can see that even extremely long latencies, such as $2^{18}$ will not destroy the efficiency, unless the latency forms a significant part of the whole execution time, as it is case if $P$ is large. Comparing the figures, we notice that even if the $B$ is more involved with $P$ in Formula (7-30) than $L$, the differences in impact of $B$ for different values of $P$ are not as large as the differences in the impact of $L$. This is because the parameter $B$ stands for the reciprocal of the bandwidth available for *each* processor.

Figure 7-12 presents a conclusive version of the combined impact of $B$ and $L$ for matrix multiplication using different numbers of processors. The graph presents a 2-dimensional space of parameters $B$ and $L$. For each $P$ there is a line to present the boundary of efficient execution compared to the optimal execution with the corresponding

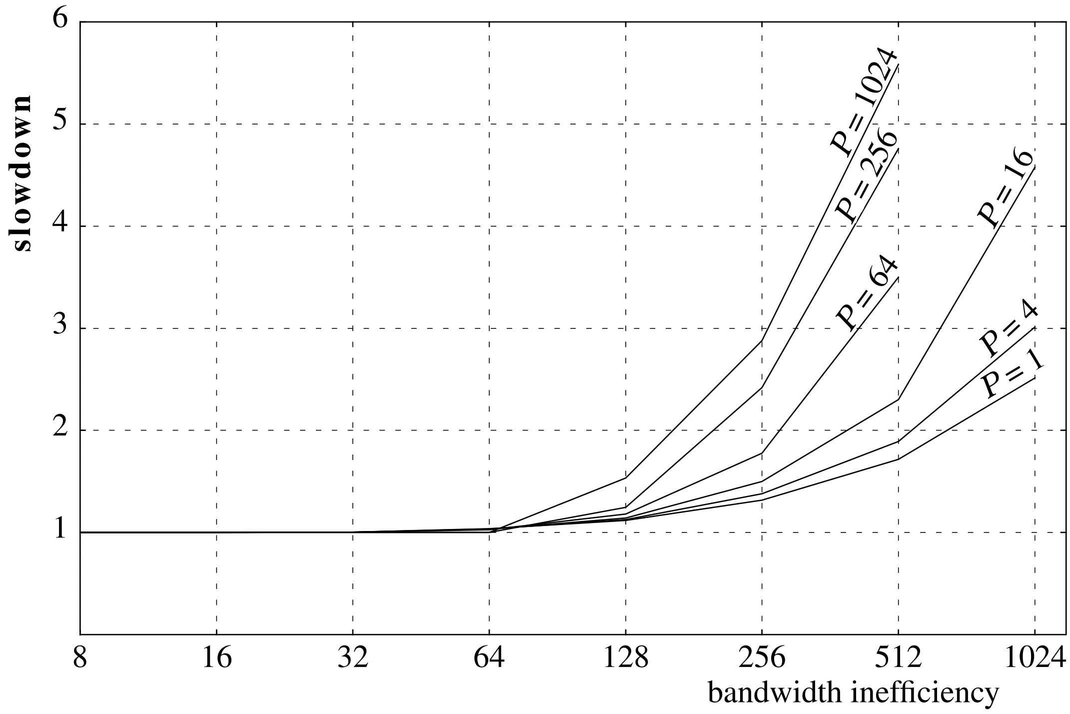


**Figure 7-10:** Slowdown of matrix multiplication as a function of bandwidth inefficiency for different numbers of processors, $N = 64$.

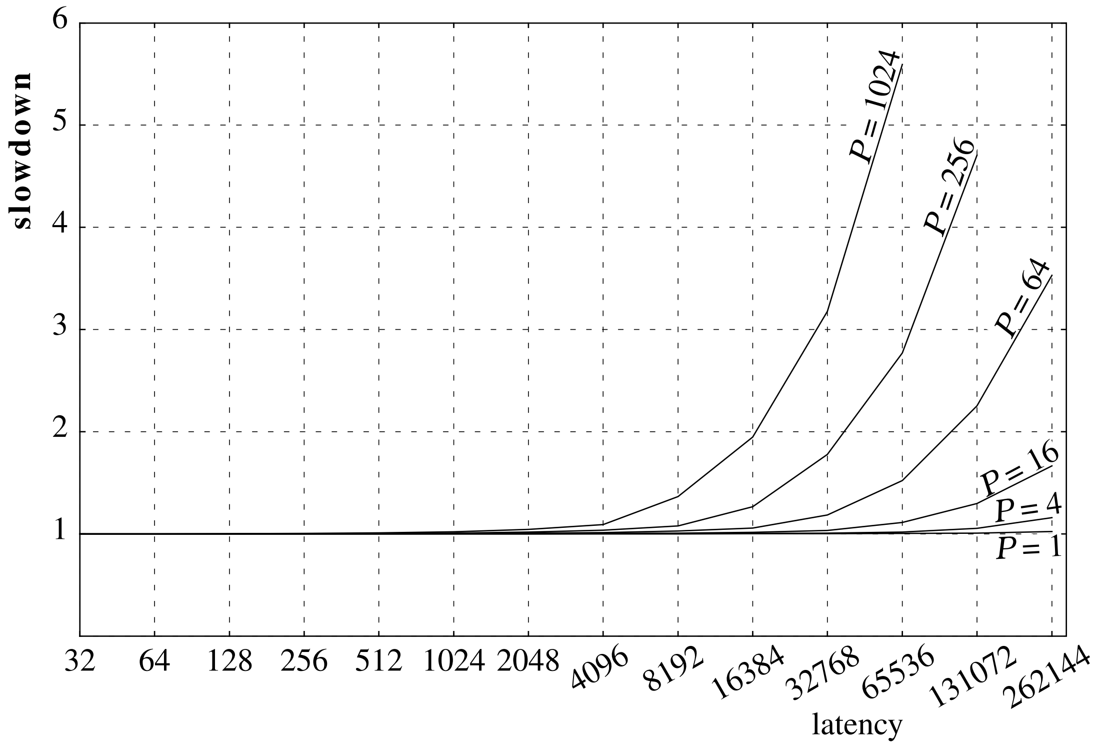


**Figure 7-11:** Slowdown of matrix multiplication as a function of latency for different numbers of processors, $N = 64$.

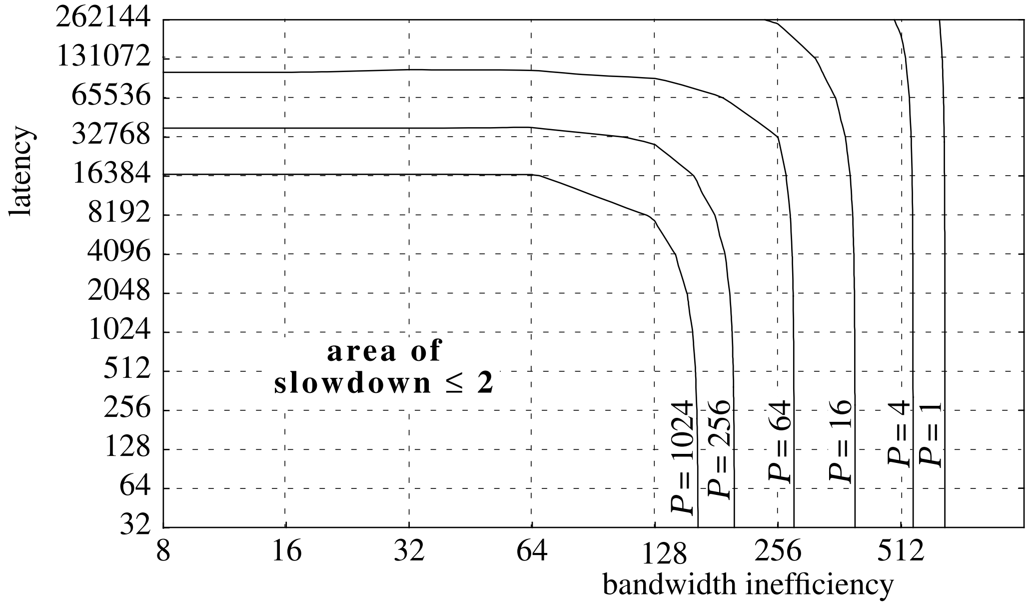


**Figure 7-12:** Acceptable (at most twice the time of the execution in an optimal machine with the same $P$) latencies and bandwidth inefficiencies in matrix multiplication for different numbers of processors, $N = 64$.

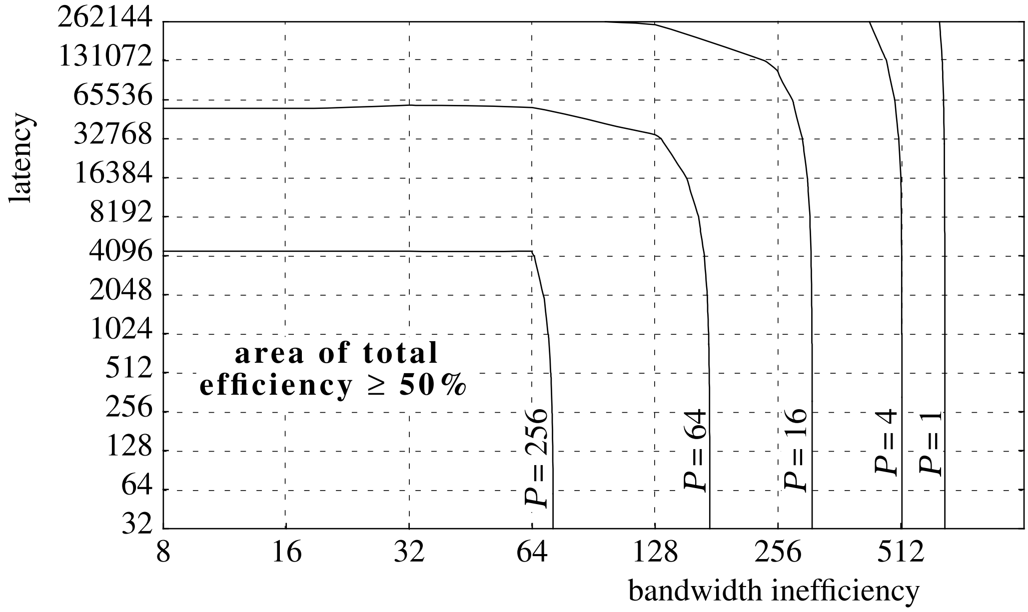


**Figure 7-13:** Requirements on bandwidth inefficiency, latency, and number of processors for efficient (at most twice the work of the sequential execution in an optimal machine) execution of matrix multiplication, $N = 64$.

$P$. The pairs of $B$ and $L$ that are below and left of the line enable execution in at most twice the time of the optimal execution for that $P$. The pairs of $B$ and $L$ that are above and right of the line induce more than 2-fold slowdown. The lines are drawn as slowdown = 2 contour lines from 3-dimensional surfaces plotted from the data for each $P$.

The graph of Figure 7-12 does not take into account the slowdown induced by the parallelization inefficiency. If we combine the results of Figures 7-9 and 7-12 we get the requirements for the total 50 % efficiency. In other words, instead of comparing each time to the time of an optimal machine with the same $P$, we compare each work to the work of an optimal machine with $P = 1$. Figure 7-13 presents this version with the same data as Figure 7-12. Now all the pairs of parameters $B$ and $L$ which are left/below a line enable us to execute to program with no more than twice the work than with a machine with $P = B = L = 1$. Note that using $P = 1024$ for this small input size, our implementation did not meet the 50 % efficiency requirement at all.

## 7.3 A larger example: matrix inversion

The inverse of an $N{\times}N$ square matrix $A$ is another $N{\times}N$ square matrix $A^{-1}$ which satisfies the equation

$$A \cdot A^{-1} = I, \qquad (7\text{-}31)$$

where $I$ is the identity matrix having 1s as diagonal elements and 0s as other elements. One way to compute the inverse $A^{-1}$ is to solve systems of linear algebraic equations

$$A \cdot x_j = b_j, \qquad j = 1..N, \qquad (7\text{-}32)$$

where $b_j$ is an identity vector with 1 in the $j$th element and 0s as other elements, and $x_j$ is the solution vector of the equation $j$. The resulting $x_j$ vectors are the columns of the inverse $A^{-1}$. For solving the systems of equations the most used methods are Gauss-Jordan elimination, Gaussian elimination with backsubstitution, and LU decomposition with backsubstitution. All of these use $O(N^3)$ arithmetic operations. As was matrix multiplication, also matrix inversion can be done with just $O(N^{\log_2 7}) \approx O(N^{2.807})$ arithmetic operations, but the asymptotically faster methods include larger constants and are much harder to program. The $O(N^3)$ algorithms are practical for many purposes. If the matrix is not dense, i.e., it contains more zeros than nonzero elements, there are more efficient methods of solving the systems of equations. Here, however, we assume that the matrix is dense.

In our example implementation we use the algorithm based on lower-upper triangular (LU) decomposition of the matrix. An LU decomposition of a matrix $A$ is a pair of matrices $L$ and $U$, which satisfy the equation

$$L \cdot U = A \qquad (7\text{-}33)$$

where $L$ is a lower triangular matrix and $U$ is an upper triangular matrix. Here the diagonal elements of the matrix $L$ are 1s. These triangular components can be used in solving systems of linear algebraic equations by writing

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b, \qquad (7\text{-}34)$$

and first solving vector $y$ from equation

$$L \cdot y = b, \tag{7-35}$$

and then solving vector $x$ from equation

$$U \cdot x = y \tag{7-36}$$

to get the final result. As the matrices $L$ and $U$ are triangular, the equations (7-35) and (7-36) can be easily computed by forward substitution and backsubstitution. The vector $y$ is computed using forward substitution by equations

$$\begin{aligned} y_1 &= \frac{b_1}{l_{11}} \ , \\ y_i &= \frac{1}{l_{ii}}\left(b_i - \sum_{j=1}^{i-1} u_{ij} x_j\right) \qquad i = 2..N \ , \end{aligned} \tag{7-37}$$

and the vector $x$ can be computed using backsubstitution by equations

$$\begin{aligned} x_i &= \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{N} l_{ij} x_j\right) \qquad i = N-1..1 \ , \\ x_N &= \frac{y_N}{u_{NN}} \ , \end{aligned} \tag{7-38}$$

where $l_{ij}$ and $u_{ij}$ are elements of matrices $L$ and $U$, respectively. To compute the $L$ and $U$ components we use Crout's algorithm with implicit pivoting [86] to form a matrix in which the elements of both $L$ and $U$ are stored in the same matrix with permuted columns.[69]

In Subsection 7.3.1 present the used algorithm in parts and analyse all parts separately, and then estimate the performance using a couple of different methods. A part of the algorithm, finding the maximum, is separately discussed in Section 7.4. In Subsection 7.3.2 we shall present the measured performance of the algorithm in the F-PRAM emulator system.

## 7.3.1 The F-PRAM implementation

A way to write parallel algorithms is to parallelize an existing sequential algorithm. In this section we present an example of modifying a sequential algorithm to a parallel F-PRAM algorithm. For this example, the sequential algorithm was taken from [86] in form of C code and rewritten to parallel FPM code. Basically, we parallelized some of the *for-do* loops of width $N$ to reduce the execution time by factor $O(P)$, $P \le N$. Even if some parts of the algorithm could have been parallelized even more, a part of the algorithm would not parallelize well enough beyond $N$ processors. Consequently, we did not parallelize any part of the algorithm beyond $N$ processors. In addition to the insertion of *par-do* state-

69. The diagonal elements of the matrix $L$ are not stored as they all are 1s.

```
procedure inverse (sharedvar a, y : matrix);
    par i := 0..N-1 do
        fwrite indx[i] := i;
    synchronize;
    LUdecomp(N, a, indx, d);
    synchronize;
    par j := 0..N-1 do
        col[0..N-1] := 0.0; col[j] := 1.0;
        LUbcksub(N, a, indx, col);
        fwrite y[0..N-1, j] := col[0..N-1];
```

**Algorithm 7-5:** Matrix inversion

ments, we had to reorganize the memory references. We chose the approach of using the shared memory for storing the matrix to be inverted to enable the communication between the processors. This causes a lot of shared memory references even if the number of processors is low, or even 1, but we feel that the algorithm would be different from the original one if we would include intermediate possibilities of using local memories. An additional excuse could be that the local memories might not be big enough to store the whole matrix at a time. To speed up the references to the arrays we have used arrays of range $[0..N-1]$ instead of the more traditional range $[1..N]$.

### Main inversion procedure

The main procedure of the inversion is very short, as it only uses the LU decomposition and backsubstitution procedures. The LU decomposition is called only once, and it contains the parallelism. The backsubstitution, however, is called $N$ times to compute the $N$ columns of the $A^{-1}$. Moreover, the evaluations of the columns are independent, and can be computed in parallel. Algorithm 7-5 presents the inversion procedure in compacted FPM. The running time of the inversion procedure is trivially

$$T_{inv}(N, P) = T_{lud}(N, P) + \lceil N/P \rceil \times T_{bcks}(N), \tag{7-39}$$

assuming that the few initialization and finishing statements are insignificant compared to the two external procedures. We shall later express the complexity more explicitly as we have analysed the terms $T_{lud}$ and $T_{bcks}$.

### LU decomposition

The F-PRAM implementation of Crout's algorithm with implicit pivoting is presented as Algorithm 7-6. The body of the procedure begins with finding maximum values from each row and storing their inverses for later scaling use (lines 3-6). This first phase can be done trivially in time

```
1  procedure LUdecomp (sharedvar a : matrix; indx : integervector; d : real);
2      fwrite d := 1.0;          imax := 0;
3      par i := 0..N-1 do
4          future T1[0..N-1] := a[i, 0..N-1];
5          big := max(T1[0..N-1]);
6          fwrite vv[i] := 1.0 / big;
7      for j := 0..N-1 do
8          fwrite tvect[0..j-1] := NotAvail;      synchronize;
9          par i := 0..j-1 do
10             future sum := a[i, j];
11             for k := 0..i-1 do
12                 future t1 := a[i, k];
13                 repeat
14                     future t2 := tvect[k];
15                 until t2 <> NotAvail;
16                 sum := sum - t1 * t2;
17             fwrite tvect[i] := sum;
18             fwrite a[i, j] := sum;
19         synchronize;      big := 0.0;
20         par i := j..N-1 do
21             future sum := a[i, j];        future dum := vv[i];
22             future T1[0..j-1] := a[i, 0..j-1];
23             future T2[0..j-1] := a[0..j-1, j];
24             sum := sum - T1[0..j-1] · T2[0..j-1];
25             fwrite a[i, j] := sum;
26             dum := dum * abs(sum);
27             if dum * abs(sum) >= big then
28                 big := dum * abs(sum); max := i;
29         sharemaxandindex(big, imax);
30         if j <> imax then
31             par k := 0..N-1 do
32                 future dum := a[imax, k];       future t1 := a[j, k];
33                 fwrite a[imax, k] := t1;        fwrite a[j, k] := dum;
34             future t1 := vv[j];                 future t2 := d;
35             fwrite vv[imax] := t1;              fwrite d := -t2;
36         synchronize;      fwrite indx[j] := imax;
37         if j <> N-1 then
38             future all t1 := a[j, j];       dum := 1.0 / t1;
39             par i := j+1..N-1 do
40                 future t1 := a[i, j];
41                 fwrite a[i, j] := dum * t1;
42         synchronize;
```

**Algorithm 7-6:** LU decomposition, the expressions around line 15 are explained in the text.

$$\lceil N/P \rceil \times \max(L, N \times \max(C_1, B)), \tag{7-40}$$

where $C_1$ is a local constant. This takes $O(N^2/P)$ time if $B = L = 1$.

The largest body of the procedure (lines 7-41) consists of an iteration over the columns of the matrix. Within the iteration are several stages which are parallelized in different ways. Most of the inner iterations have range $0..j-1$ or $j+1..N-1$, where $j$ is the current column of the outer iteration. In the following analysis we shall use the variable $j$ to denote this.

The first part (lines 9-18) of the iteration is to find the elements of $U$ by equation

$$u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik} u_{kj}, \tag{7-41}$$

which could be easily parallelized but unfortunately the expression uses the previous values of $u_{kj}$ to compute the new $u_{ij}$. This does not prevent parallelization but we have to carefully ensure that the new values of $u_{kj}$ are used in the computation. In other words, we pipeline the execution to use the value of a new $u_{kj}$ as soon as it is computed. To accomplish this, we use a separate temporary shared array of the new values of $u_{kj}$. Before this stage, the array is filled with special *not-yet-available* (*NotAvail*) values, which are replaced by the correct one as soon as a processor is able to compute one. Instead of referencing directly the matrix to obtain the $u_{kj}$ values, the processors use the temporary array. If the reference to the temporary array returns the *NotAvail* value, the processor tries again until it gets a valid value. If $P = N$, the pipelining doubles the time needed, but if $P < N$, the impact is smaller, and if $P = 1$, there is no additional wait. More formally, the time needed for this stage is

$$\lceil j/P \rceil \times (j + P) \times \max(L, \max(B, C_2))), \tag{7-42}$$

which is not very good if the latency $L$ of the shared memory access is large. In fact this stage forms the longest communication path of the whole algorithm, and will be the slowest of the whole inversion for large values of $L$. This stage takes $O(N^2/P)$ time if $B = L = 1$.

The next stage (lines 19-29) of the iteration finds an element of the remaining matrix to be chosen for the next pivot element. The finding the maximum is parallelized by making all processors find the maximum independently from their own slices as in the sequential version, and afterwards find and distribute the maximum over all processors. The parallelized part takes time

$$\lceil (N-j)/P \rceil \times j \times \max(L, \max(B, C_3))). \tag{7-43}$$

To share the found maximums among the processors, we use a separate routine *sharemaxandindex*, which we shall describe and analyse separately in Section 7.4. Here it suffices to state that the procedure to find the maximum and the corresponding index over $P$ processors takes time

$$\max(L, d\times\max(B, C_m))\times(\lceil \log_d P\rceil + 2), \tag{7-44}$$

where $d$ is a variable assigned according to $L$ and $B$. In any case, the maximum over the processors is fast compared to the initial work done by the processors independently. Using $O$-notation an iteration of the whole finding the maximum stage takes $O(N^2/P)$ time.

The next stage (lines 30-36) of the iteration is to possibly interchange rows to get the pivot element on the diagonal. This operation can be done in time

$$\lceil N/P\rceil\times\max(L, \max(B, C_4))) = O(N/P), \quad \text{if } B = L = 1, \tag{7-45}$$

i.e., this is a faster operation than the previous ones.

The final stage (lines 37-41) of the iteration is to do the division by the pivot element in time

$$\lceil (N-j)/P\rceil\times\max(L, \max(B, C_5))) = O(N/P), \quad \text{if } B = L = 1. \tag{7-46}$$

The time of the whole iteration $j := 0$ to $N-1$ contains the equations (7-42), (7-43), (7-44), (7-45), and (7-46) over all values of $j$. The costs of equations (7-42), (7-43), and (7-46) involve $j$, and are combined to a single sum

$$\left(\sum_{j=0}^{N-1}\left\lceil \frac{j}{P}\right\rceil\times(2j+P+1)\right)\times\max(L, \max(B, C_{lu})), \tag{7-47}$$

where $C_{lu}$ is the maximum of the constants $C_2$, $C_3$, and $C_5$. By combining this to formulas (7-44) and (7-45) we get

$$T_{lud} = \left(\left(\sum_{j=0}^{N}\left\lceil \frac{j}{P}\right\rceil\times(N+P+1)\right)+N\log P\right)\times\max(L, \max(B, C_{lu})) \tag{7-48}$$

as the whole complexity of the LU decomposition.

### Forward and backsubstitution

As we stated earlier, the procedure of forward and backsubstitution is called for all columns in parallel. Therefore, within the computation of one column we proceed sequentially. Algorithm 7-7 presents the stages given in equations (7-37) and (7-38) in FPM. For each column, both forward and backsubstitution stages take at most time

$$\sum_{i=0}^{N-1}\max(L, i\times\max(B, C_{lufb})). \tag{7-49}$$

The whole procedure thus takes time

```
procedure LUbcksub (sharedvar a : matrix; sharedvar indx : ivector;
                    var b : vector);
    ii := -1;
    future IP[0..N-1] := indx[0..N-1];
    for i := 0..N-1 do
        ip := IP[i];        sum := b[ip];      b[ip] := b[i];
        if ii <> -1 then
            future T1[ii..i-1] := a[i, ii..i-1];
            sum := b[ip] - T1[ii..i-1] · b[ii..i-1];
        else
            if sum <> 0.0 then    ii := i;
        b[i] := sum;
    for i := N-1..0 by -1 do
        future t1 := a[i, i];
        future T1[i+1..N-1] := a[i, i+1..N-1];
        sum := b[i] - T1[i+1..N-1] · b[i+1..N-1];
        b[i] := sum / t1;
```

**Algorithm 7-7:** LU forward and backsubstitution.

$$T_{bcks} = N \times \max(L, N \times \max(B, C_{lufb})) = O(N^2), \quad \text{if } B = L = 1. \tag{7-50}$$

**Combined analysis of the whole matrix inversion**

Combining the equations (7-39), (7-48), and (7-50) we get the running time of the matrix inversion as follows

$$T = \left(\left(\sum_{j=0}^{N} \left\lceil \frac{j}{P} \right\rceil \times (N + P + 1)\right) + N \log P\right) \times \max(L, \max(B, C_{lu})) + \tag{7-51}$$
$$\left\lceil \frac{N}{P} \right\rceil \times N \times \max(L, N \times \max(B, C_{lufb}))$$

$$\leq \left(\left(\frac{N}{2} \times \left\lceil \frac{N}{P} \right\rceil \times (N + P + 1)\right) + N \log P\right) \times \max(L, \max(B, C_{lu})) + \tag{7-52}$$
$$\left\lceil \frac{N}{P} \right\rceil \times N \times \max(L, N \times \max(B, C_{lufb}))$$

$$\leq \frac{N^2}{P} \times N + 2 \times \max(L, \max(B, C_{lu})) + \tag{7-53}$$

$$\max(L, N \times \max(B, C_{lufb}))$$

$$\leq \frac{2N}{P} \times (N+1) \times \max(L, \max(B, N \times C_{lu})) \tag{7-54}$$

which differs from the optimal time only by a constant factor 2 if $B = L = 1$. Examining the earlier stages of the analysis, we note that for small values of $P$ the efficiency is better than if $P$ is close to $N$. Especially values of $P$ in range $N/2+1$ .. $N-1$ do not give much additional speedup. Using the $O$-notation, however, we get the desired time complexity

$$T_{inv} = O\left(\frac{N^3}{P}\right), \quad \text{if } P \leq N \text{ and } B = L = 1. \tag{7-55}$$

The impacts of $L$ and $B$, however are linear after some limit if $P = N$. Making even moderately accurate estimations of the actual speedup performance of the algorithm is difficult using any of the above formulas, though Figure 7-14 includes a plot of Formula (7-51).

**Another approach to speedup estimation**

As we noticed above, the analysis of the potential speedup may be a bit difficult using the traditional analysis techniques. Especially the meaning of the constants may be unclear. As we have implemented the algorithm for a working emulator system, we can gather performance information from an actual execution. For example, we can modify the program to print the width of the range of every *par-do* statement it executes. Then for each possible $P$, we can compute the number of iterations a processor needs to do in each *par-do* statement. For the matrix inversion program we get 4 occurrences[70] of *par-do* statements of width $i$, $i \in 0..N-1$, and $N$ occurrences of *par-do* statements of width $N$. Moreover, the procedure to find the maximum does not contain any *par-do* statement, yet it works in parallel. Consequently, the number of *par-do* body iterations would be

$$4\sum_{i=0}^{N-1}\left\lceil\frac{i}{P}\right\rceil + N\left\lceil\frac{N}{P}\right\rceil + N\log P, \tag{7-56}$$

but that formula does not take into account the fact that one of the width $N$ *par-do* statements includes an $O(N^2)$ body, whereas the other par-do statements include only $O(N)$ bodies. Further, two of the width $i$ *par-do* statements and the finding the maximum iteration include $O(1)$ bodies, whereas the rest include $O(N)$ bodies. Thus, we adjust Formula (7-56) to form

$$2\sum_{i=0}^{N-1}\left\lceil\frac{i}{P}\right\rceil + 2N\left\lceil\frac{N}{P}\right\rceil + \log P, \tag{7-57}$$

which can be converted to the total complexity of the inversion by multiplying it by the complexity of the bodies of the *par-do* statements, i.e., $N$. The interesting fact of Formula (7-57) is the strong impact of the roof operations, which was hidden in the $O$-anal-

70. One simple *par-do* statement is not visible in the compacted algorithms.

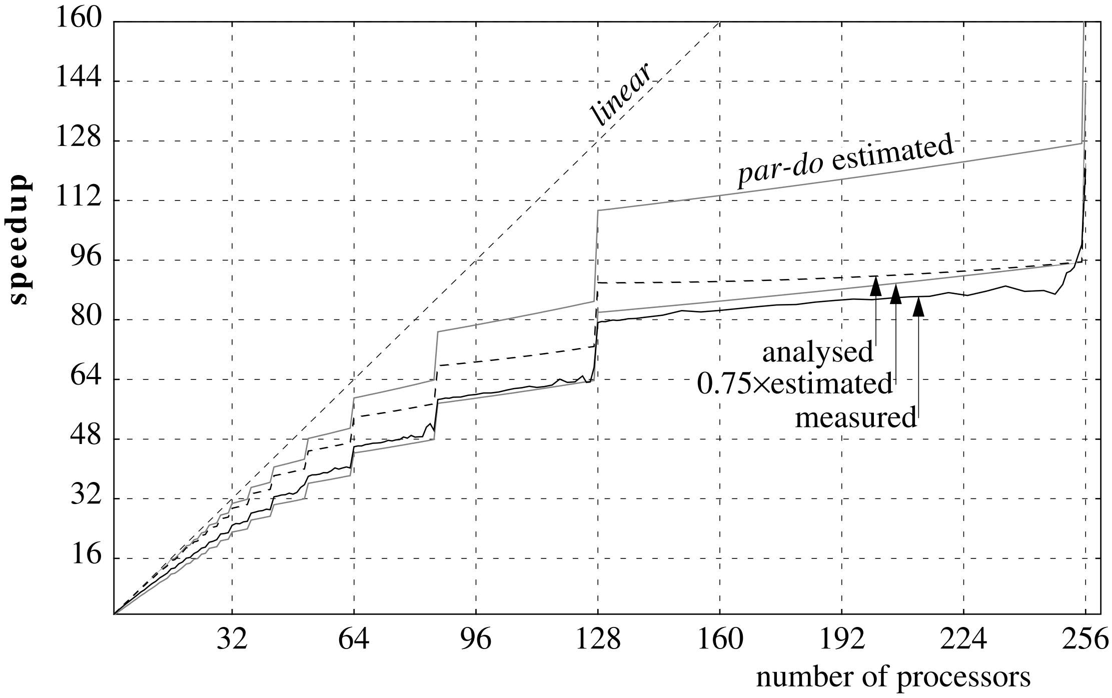


**Figure 7-14:** Speedups calculated from the measured execution of the matrix inversion, analysed (7-51), *par-do* estimated (7-57), and scaled *par-do* estimated times of the matrix inversion, $N = 256$.

ysis. Using the formula, we can plot the potential speedup of the matrix inversion. Figure 7-14 includes the plots of speedups computed using Formulas (7-57) and (7-51), and the measured performance described in the next subsection. The *par-do* estimated version does not take into account the fact that sequential execution of the *par-do* is a bit more efficient than parallel execution due to the reduced impact of initialization costs. As the estimates give too slow $P = 1$ estimate, they result in too good speedup figures. To correct this estimation error, the graph also includes a fitted (scaled by factor 0.75) version of the speedup curve of Formula (7-51). This fitted version follows surprisingly close to the actual measured graph. The stair-shape of the graphs is due to the parallelization technique used in backsubstitution phase. For example, 256 distinct backsubstitutions can be done considerably faster with 64 processors than with 63 processors. Still, adding a 65th processor does not help backsubstitution phase at all. It does, however, help the LU decomposition phase.

### 7.3.2 Measured performance

Again we used mostly the same assumptions as before. However, the algorithm uses the shared memory even if the machine had only one processor. This can be considered unfair, especially as this algorithm uses quite a lot of shared memory. This algorithm was, however, an example of a straightforward parallelization of a sequential algorithm. Figure 7-15 presents the speedup as a function of the number of processors. Different lines present the speedup curves measured with different input sizes. The input $N$ stands for inverting an $N \times N$ matrix. The speedups are measured relative to the execution of the

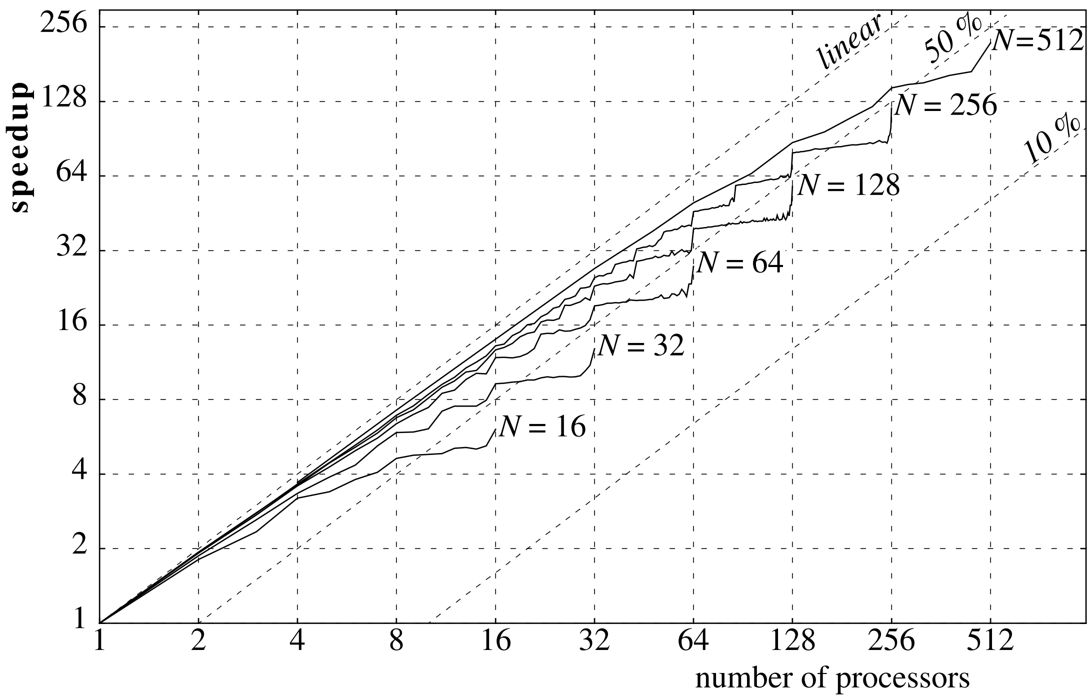


**Figure 7-15:** Speedup of the matrix inversion as a function of the number of processors for different input sizes.

same algorithm using only one processor, except for $N = 512$, which could not be executed with $P \leq 4$ with our current emulator.[71] Consequently, for $N = 512$, the speedups are based on the execution with $P = 4$, with the base speedup of $P = 4$ taken from the case $N = 256$. Moreover, as opposed to the smaller input sizes, for $N = 512$, we did not run the experiments for every value of $P$. The steps of the graphs show us the obvious fact that even fractions of $N$ are the best values for $P$.

As we may guess from the analysis of Algorithm 7-5, the two stages of the inversion are approximately equally time-consuming. The parallelization of the stages, is not, however, similar because the backsubstitution stage mostly includes only one width-$N$ *par-do* statement. Figure 7-16 presents the relative speedups of the two stages of the inversion algorithm. As we can see, the backsubstitution stage parallelizes better, but achieves the best efficiencies at factors of $N$. As the LU decomposition mostly includes *par-do* statements with range $j, j \in 1..N$, it gains from the added processors more evenly. The variation on the curve of the backsubstitution is due the random variation on the input data.

### The impact of latency and bandwidth inefficiency

Considering the impact of the shared memory access parameters, we ran the same 64×64 matrix inversion varying the parameters $P$, $B$, and $L$ forming a 3-dimensional domain of executions. The following graphs present the results of these experiments. Figure 7-17 presents the impact of bandwidth inefficiency $B$ for different values of $P$ with low $L$. The bandwidth inefficiency starts to impact beyond $B = 32$, and impacts significantly thereaf-

71. The number of clock cycles needed for the inversion was larger than the maxint $2^{31}$.

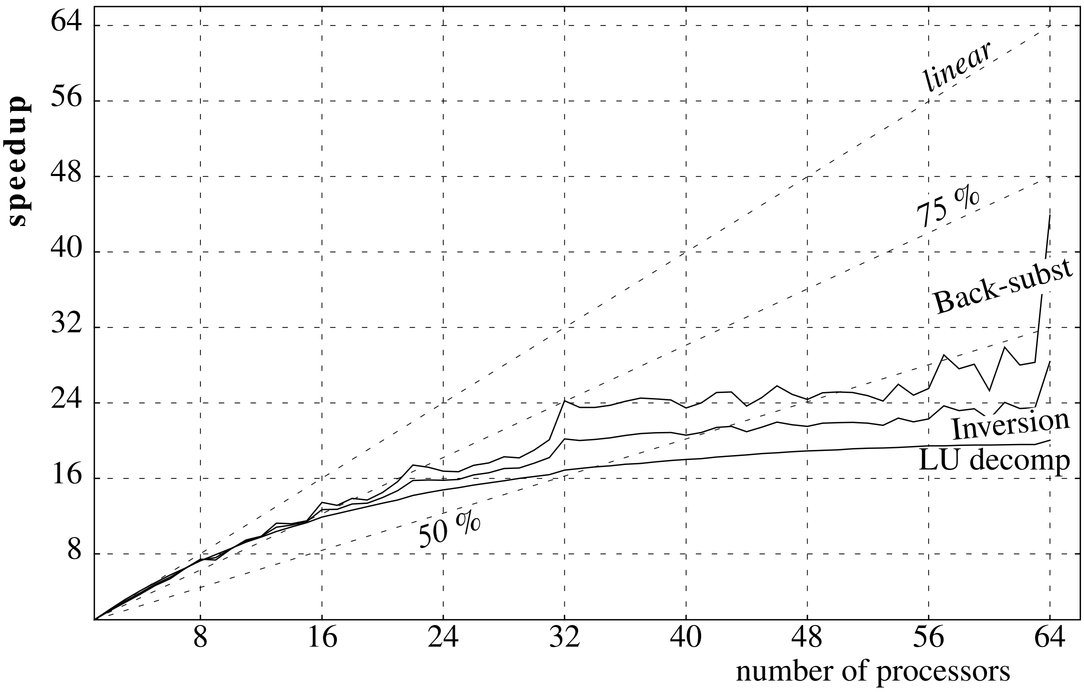


**Figure 7-16:** Speedup of matrix inversion, LU decomposition, and LU back-substitution as functions of the number of processors, $N = 64$.

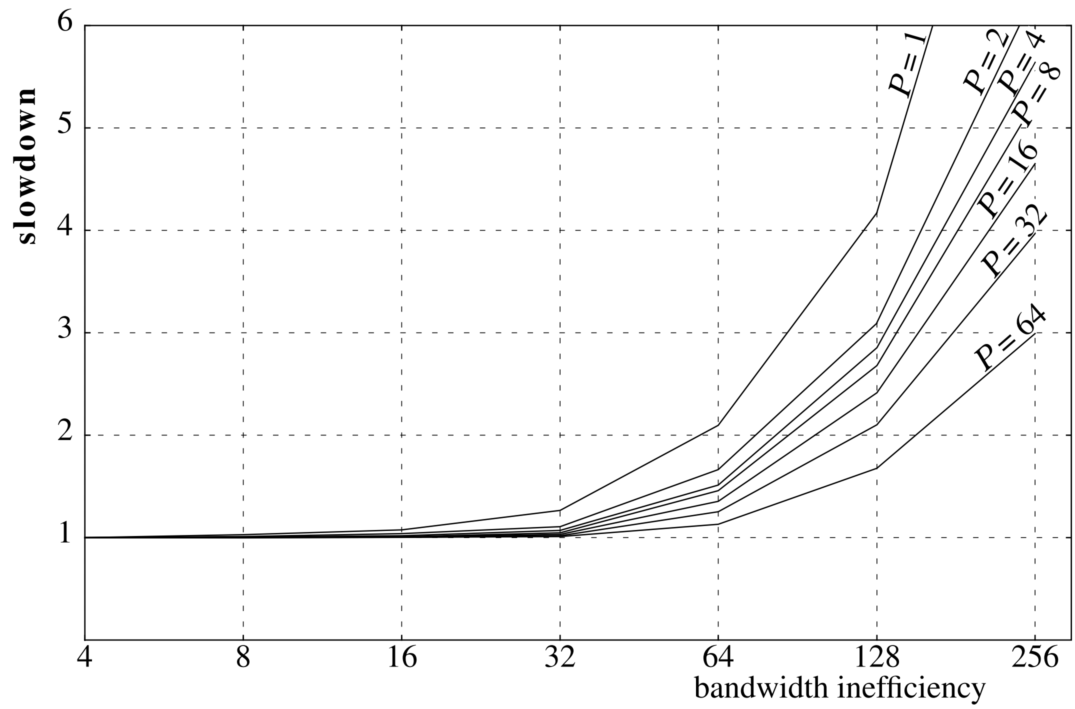


**Figure 7-17:** Slowdown of the matrix inversion as a function of bandwidth inefficiency for different numbers of processors, $N = 64$.

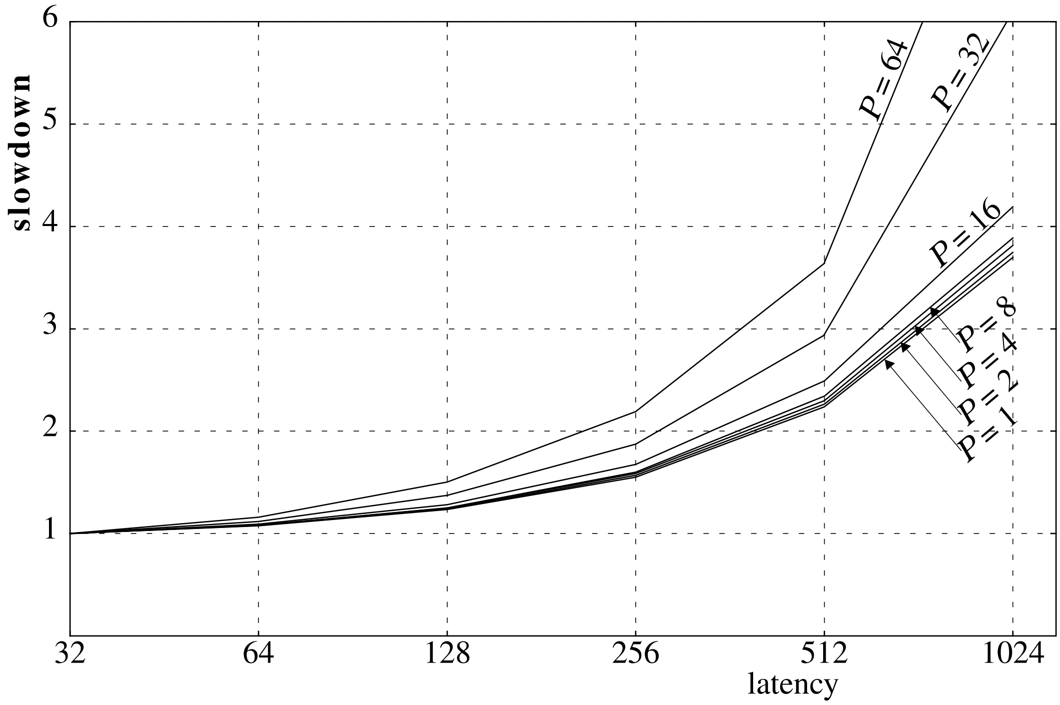


**Figure 7-18:** Slowdown of the matrix inversion as a function of latency for different numbers of processors, $N = 64$.

ter. Notice, that the smaller is $P$, the larger is the impact of $B$. This is the very opposite effect compared to any other algorithm we have presented this far. The basic reason why even an execution with low $P$ does not tolerate much $B$ is because the matrix to be inverted is kept all the time in the shared memory, and, thus, the processors need to access it constantly. This does not explain the fact that a machine with smaller $P$ would require a better (lower) value of $B$ than a machine with a high $P$. The volume of shared memory accesses remains constant with respect to work done. Consequently, as the amount of shared memory bandwidth and the work done in a time unit increase at the same rate as the number of processors increase, the requirement for $B$ should remain constant. However, because the relative efficiency decreases as $P$ increases, the available shared memory access bandwidth increases faster than the work done. Particularly, as some of the processors do not do useful work, the rest of the processors may exploit the bandwidth of the idling processors. Later, in Figure 7-20 we take this into account, and find that the bandwidth requirement really is constant with respect to the work done. Compared to matrix multiplication, Figure 7-10, the impact of $B$ with small values of $P$ is significant in the inversion algorithm.

Figure 7-18 presents the impact of latency $L$ for different values of $P$ with low $B$. The latency impacts already with very low values. This is because the longest communication path of the LU decomposition is $N^2$. Thus, the algorithm takes at least time $N^2L$. More specifically, on line 15 of Algorithm 7-6 the processors wait until the previous value has arrived. As the processors are actively waiting for the value, even very low latencies impact the performance. For a low $P$ the impact is lower as the processors have other iterations to complete before the waiting occurs. Compared to matrix multiplication (similar

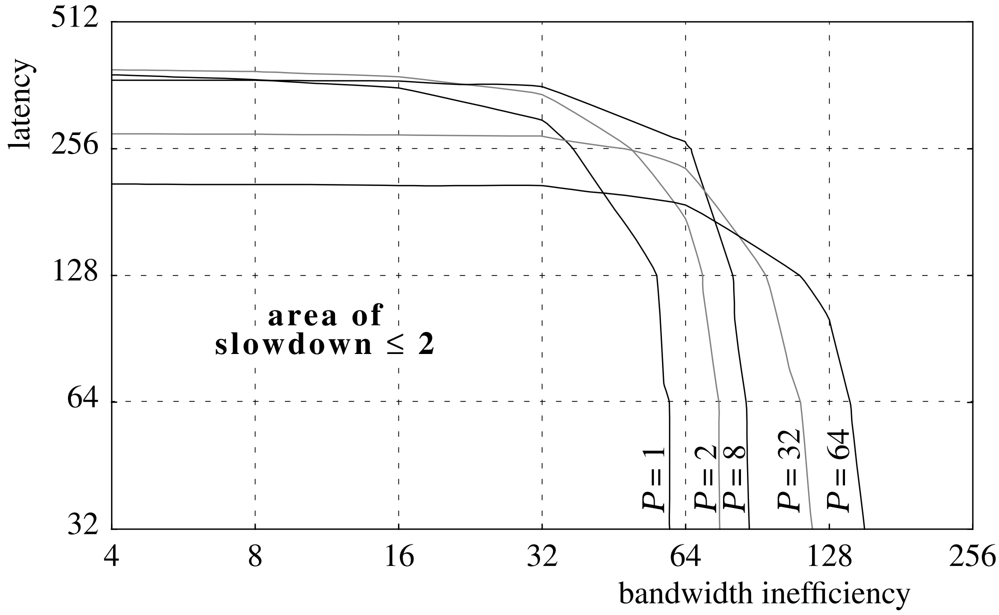


**Figure 7-19:** Acceptable (at most twice the time of the execution in an optimal machine with the same $P$) latencies and bandwidth inefficiencies in matrix inversion for different numbers of processors, $N = 64$.

graph in Figure 7-11), the difference of latency toleration is a couple of orders of magnitude worse.

Figure 7-19 presents a conclusive graph of the combined impact of $B$ and $L$ for the matrix inversion using different numbers of processors. The graph presents a 2-dimensional space of parameters $B$ and $L$. For each $P$ there is a line to present the boundary of efficient execution compared to the optimal execution with the corresponding $P$. The pairs of $B$ and $L$ that are below and left of the line enable execution in at most twice the time of the optimal execution for that $P$. The pairs of $B$ and $L$ that are above and right of the line induce more than 2-fold slowdown.

The graph of Figure 7-19 does not take into account the slowdown induced by the parallelization inefficiency. If we combine the results of Figures 7-16 and 7-19, we get the requirements for the total 50 % efficiency. As before, instead of comparing each time to the time of an optimal machine with the same $P$, we compare each work to the work of an optimal machine with $P = 1$. Figure 7-20 presents this version with the same data as Figure 7-19. Now all the pairs of parameters $B$ and $L$ which are to the left and below of a line enable us to execute the program with no more than twice the work than with a machine with $P = B = L = 1$. Note that using $P > 40$, our implementation did not meet the 50 % efficiency requirement at all. As an interesting fact we can notice that most of the lines of Figure 7-20 meet at $B \approx 80$, $L \leq 64$. In other words, to be able to efficiently invert a matrix using our implementation, we need a machine with $B \leq 64$. We already discussed the bandwidth requirements earlier within the measurements concerning $B$ and $P$.

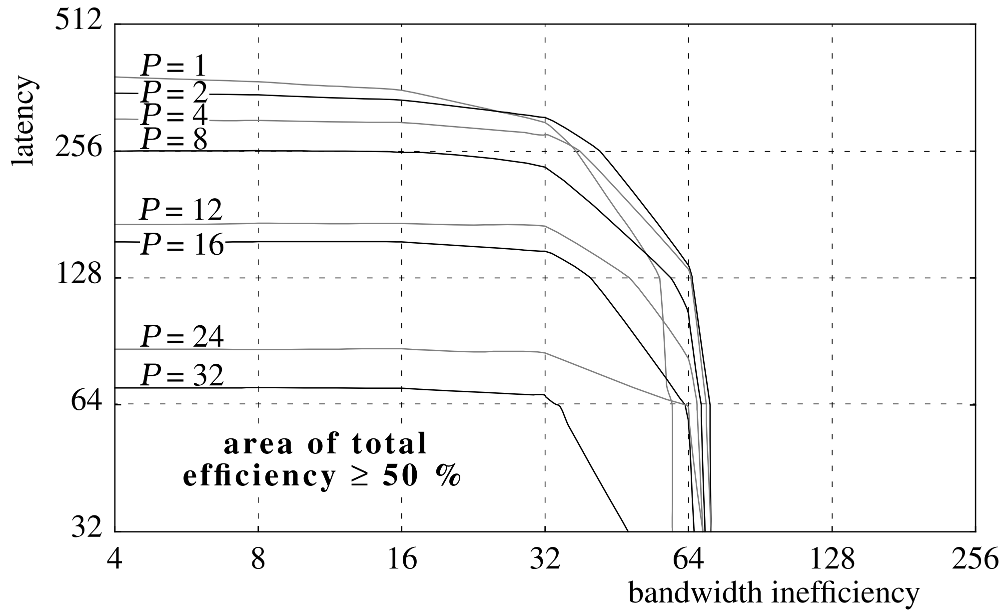


**Figure 7-20:** Requirements on bandwidth inefficiency, latency, and number of processors for efficient (at most twice the work of the sequential execution in an optimal machine) execution of matrix inversion, $N = 64$.

## 7.4 Maximum over processors

The LU decomposition procedure (Algorithm 7-6) finds the maximum of a vector of inner product results and the corresponding index (lines 20-28) in parallel. The result is that each processor has its own maximum and corresponding index over the range it processed. To achieve a common maximum for all processors, we use a separate routine *sharemaxandindex* (line 29). Here we shall study the routine separately as the finding the maximum is generally considered as a distinct problem and algorithm.

The routine used by the LU decomposition takes two variable parameters, the value, and the index. Both actual parameters must be local variables of the processors. At the exit of the procedure, the parameter *value* of each of the processors will equal to the maximum of the original values over all processors. The parameter *index* will equal to the corresponding index.

The standard algorithm for finding a maximum of a vector, or the maximum of local variables over a set of processors would be from a binary tree communication among the processors. The execution would take $O(L \times \log_2 P)$ time. To reduce the impact of the latency, we can exploit a tree with a degree higher than 2 by finding a maximum of more than 2 elements in each iteration. In general the height of the tree would be $\lceil \log_d P \rceil$, where $d$ is the degree of the tree. Thus, the execution time would be $O(\max(L, d) \times \log_d P)$. As we are using the exclusive read memory model, also the result of the maximum would have to be distributed in a $d$-ary tree manner. Moreover, we have to find and distribute the index of the maximum value, which might require a third stage of $d$-ary tree communication.

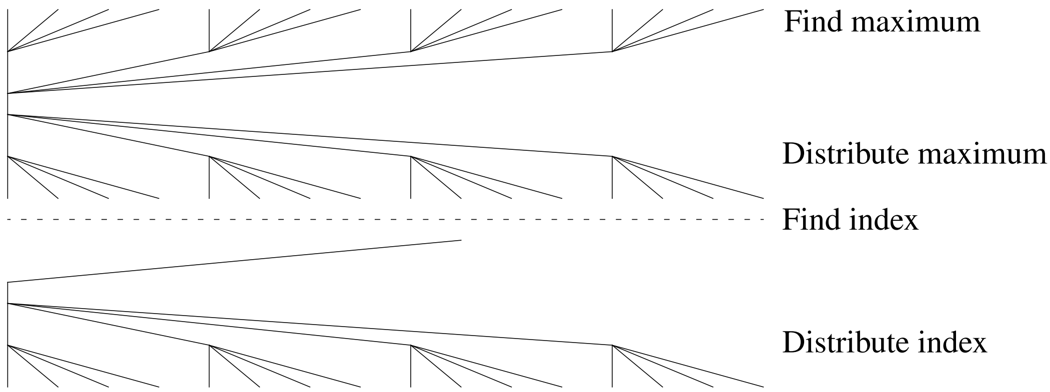


**Figure 7-21:** Three 4-ary trees required to find maximum and index using a straightforward algorithm.

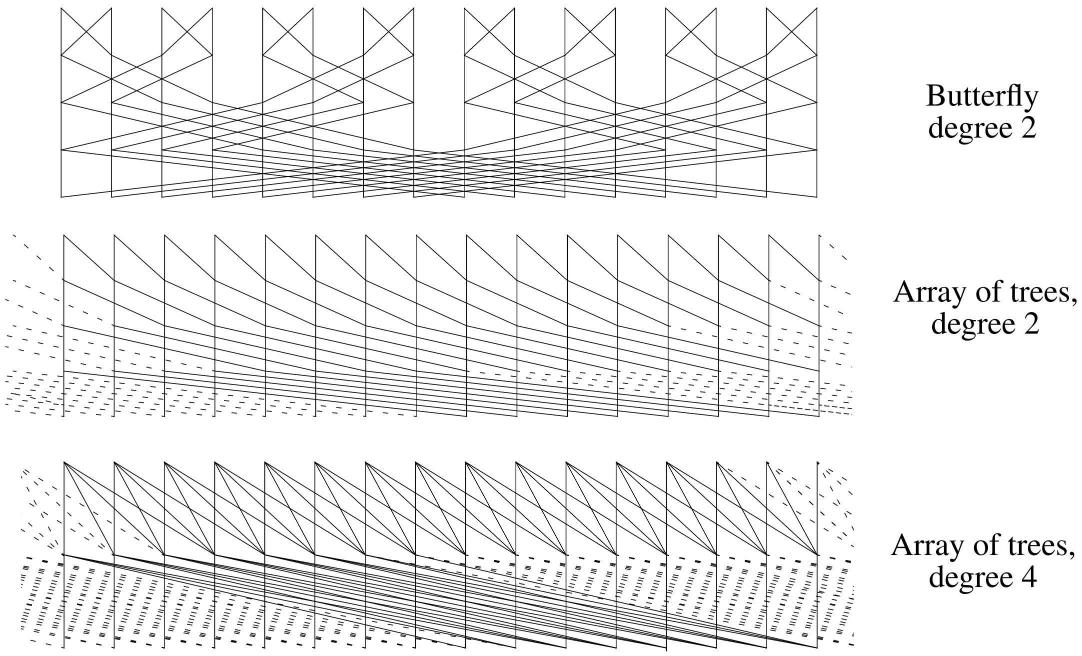


**Figure 7-22:** A butterfly network of degree 2 and "arrays of trees" of degrees 2 and 4. Dashed lines represent wrap-around edges.

Figure 7-21 presents the stages of communication between the processors using a 4-ary tree. The factor 3 in the time requirement is rather significant in a frequently used routine as the finding the maximum Moreover, most of the processors only do constant work, and only one works all the time. Since we are discussing finding the maximum value over a set of processors, we cannot improve the efficiency by blocking the original vector and reducing the number of processors. Instead, we should exploit all processors as efficiently as possible to gain the minimum possible total time.

To exploit all processors efficiently, and to get rid of the distribution stages, we virtually form a binary tree for each processor, i.e., each processor acts like it would be the root processor of a binary tree structure. If we draw this communication, we get the middle pattern of Figure 7-22. If we interpret the pattern as a graph, we formalize it by stating

```
procedure maxandindex(var value : real; var index : word);
    fwrite maxarray[PID(0)] := value;
    fwrite maxindex[PID(0)] := index;
    shift := 1;
    synchronize;
    while shift < P(0) do
        future T1[1..deg-1] := maxarray[(shift * 1..deg-1 + PID(0)) mod P(0)];
        future T2[1..deg-1] := maxindex[(shift * 1..deg-1 + PID(0)) mod P(0)];
        for i := 1 to deg-1 do
            if (T1[i] > value) or ((T1[i] = value) and (T2[i] < index)) then
                value := T1[i];   fwrite maxarray[PID(0)] := value;
                index := T2[i];   fwrite maxindex[PID(0)] := index;
        shift := shift * deg;
        synchronize;
```

**Algorithm 7-8:** Finding maximum and index over the processors using a *deg*-ary array of trees.

that it has $\lceil \log_2 N \rceil + 1$ levels of $N$ nodes each. We present each node by tuple $<l, s>$, where $l$ stands for the level and $s$ stands for the node within the level. Nodes $<l, s>$ and $<l{+}1, t>$, where $l \in 0..\lceil \log_2 N \rceil$ and $s, t \in 0..N-1$, are connected if, and only if, $t = s$ or $t = (s+2^l) \bmod N$. For $P = 2^n$, the pattern is quite similar to an $n$-dimensional butterfly, as in the topmost pattern of Figure 7-22. The pattern is so obvious that it probably has a proper name, but unfortunately we could not find a reference for it. Here we shall call it an *array of trees*.

In algorithm to find the maximum, each processor compares its own value with the value of processor $(PID+2^i) \bmod P$, where $i \in 1..\lceil \log_2 P \rceil$ is the iteration round. The advantage of this approach is that the indexing works even if $P$ is not a power of 2, and is easier to index in a practical implementation. Using this communication pattern each processor has the correct maximum value after $\lceil \log_2 P \rceil$ iterations. Moreover, because we can distribute the index along the maximum, we can reduce the number of iterations to one.[72] To further reduce the length of the longest communication path, we can exploit the trick of using an array of $d$-ary trees, as in the undermost pattern of Figure 7-22.

The whole algorithm to find the maximum is presented as compacted source code in Algorithm 7-8. The variable *deg* in the algorithm stands for the degree $d$ of the array of trees. The whole execution time of the algorithm is

$$T(P) = \max(L, C_m \times d) \times (\lceil \log_d P \rceil + 2), \qquad (7\text{-}58)$$

where $C_m$ is a local constant, assuming that we have enough bandwidth. Taking $B$ into account, the time is

72. We could have used this technique also with the original tree-based algorithm, which would had reduced the number of trees to two.

$$T(P) = \max(L, d \times \max(B, C_m)) \times (\lceil \log_d P \rceil + 2), \tag{7-59}$$

where $C_{ma}$ is a local constant. The slower tree-based approach could be slightly better in a machine with large $B$, as the volume of communication would be slightly reduced. On iteration $i$, the version based on the array of trees uses on average $3 \times d \times P$ words of bandwidth, whereas the tree-based one uses only $3 \times d \times d^{\log_d P - i}$ words of bandwidth. The tree-based algorithm requires at least two $1..d$ iterations. Thus, the whole tree-based algorithm requires

$$6 \times d \times \sum_{i=1}^{d} d^{\log_d P - i} \leq 12 \times d \times P \tag{7-60}$$

words of bandwidth. The array of trees algorithm requires

$$3 \times d \times P \times \log_d P \tag{7-61}$$

words of bandwidth. Consequently, the ratio of total bandwidth usage is

$$\frac{3 \times P \times d \times \log_d P}{12 \times P \times d} = \frac{\log_d P}{4}, \tag{7-62}$$

which, e.g., when $P = 1024$ and $d = 4$ is 1.25. As a conclusion, we can state that the bandwidth usages will not differ significantly unless the $B$ and $P$ are very large and $L$ is small.

As a conclusion of Formula (7-59) we can tell that increasing $d$ will reduce the impact of $L$ but it will also increase the impact of $B$. Using the formulas (7-58) and (7-59), we should be able to decide a convenient formula for assigning the variable *deg* in Algorithm 7-8, but unfortunately we cannot find an easy one which could be computed in the program with a reasonable amount of work. At least the processors should do something useful during $L$. In other words, we should try to set

$$d = \left\lfloor \frac{L}{\max(2B, C_m)} \right\rfloor, \tag{7-63}$$

where the multiplier 2 is due to the possible rewriting to the shared memory. The formula does not give the optimal value, but at least makes the processors to do something useful. Table 7-1 presents the computed values of $d$ for some values of $B$ and $L$ when $P = 256$ and $C_m = 67$. As we ignored here the impact of term $\lceil \log_d P \rceil$ in Formula (7-59), the values for $d$ given in the table are a bit too small. We can check this later on when comparing the table to the values in the measured Table 7-3. Further pursuing for the most appropriate values of $d$, the most efficient values are those which form a set of full trees without performing extra work. More formally, $d$ should conform to equation

**Table 7-1:** Estimated (using Formula 7-63) best values of $d$ in finding the maximum for different values of $L$ and $B$, $P = 256$, $C_m = 67$.

| Latency | Bandwidth inefficiency $B$ | | | | | | |
|---|---|---|---|---|---|---|---|
| $L$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| 8 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 16 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 32 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 64 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 128 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 256 | 3 | 3 | 3 | 3 | 2 | 2 | 2 |
| 512 | 7 | 7 | 7 | 7 | 4 | 2 | 2 |
| 1024 | 15 | 15 | 15 | 15 | 8 | 4 | 2 |
| 2048 | 30 | 30 | 30 | 30 | 16 | 8 | 4 |
| 4096 | 61 | 61 | 61 | 61 | 32 | 16 | 8 |
| 8192 | 122 | 122 | 122 | 122 | 64 | 32 | 16 |
| 16384 | 244 | 244 | 244 | 244 | 128 | 64 | 32 |

**Table 7-2:** The number of steps in the finding the maximum for different values of $d$, $P = 256$.

| degree $d$ (= steps/iteration) | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|---|
| number of iterations | 8 | 4 | 3 | 2 | 2 | 2 | 2 | 1 |
| number of total steps | 16 | 16 | 24 | 32 | 64 | 128 | 256 | 256 |

$$d^{\lceil \log_d P \rceil} = P \qquad (7\text{-}64)$$

to achieve full efficiency. If $P = 2^p$, then $d$ should be of the form $2^x$, where $x$ is a factor of $p$. For $P = 256$ the practical values are 2, 4, 16, and 256, which, as we can see later in Figure 7-25, cover the range of $L$ rather well. The problem is that using Formula (7-64) in an actual program can be easily done only using an if-elsif-...-else sequence, which is neither a general, nor an elegant solution.

Yet another way of studying the use of $d$ would be to count the number of steps each processor needs to compute. The number of comparison steps is about $(d-1) \times \lceil \log_d P \rceil$, which can be rounded to $d \times \lceil \log_d P \rceil$ since starting and finishing each iteration takes time. Table 7-2 presents the values for $d = 2^n$, $\lceil \log_d P \rceil$, and $d \times \lceil \log_d P \rceil$. We can see that degrees such as 32, 64, and 128 are of little use for $P = 256$ since they do not reduce the number of iterations, i.e., the impact of $L$, but increase the number of steps on each iteration.

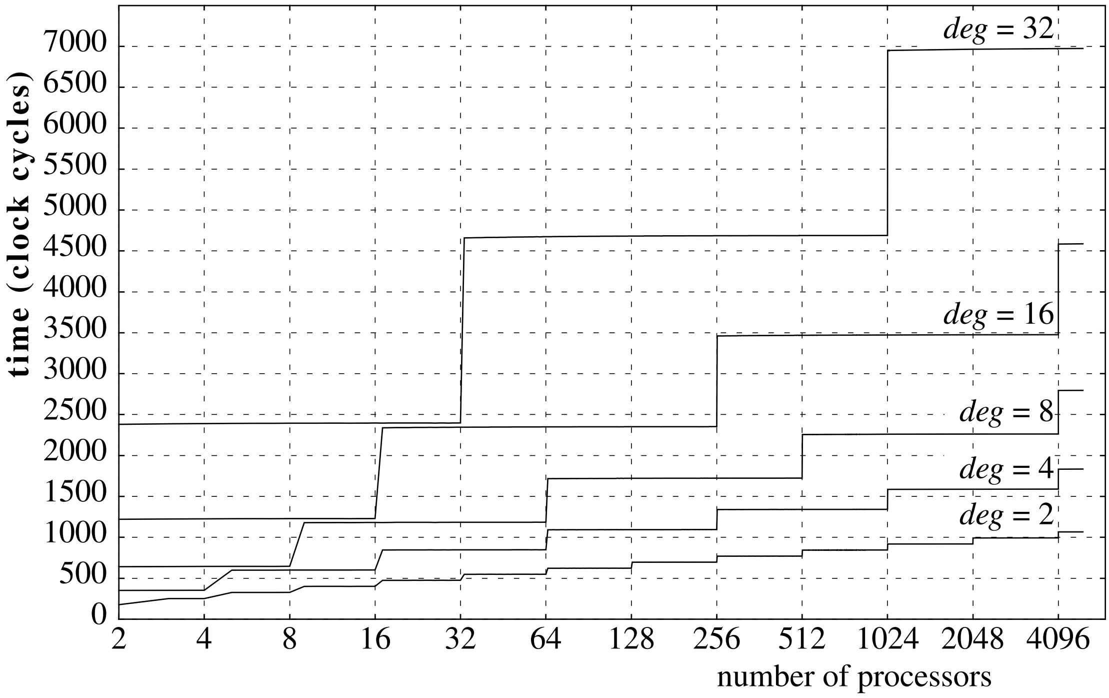


**Figure 7-23:** Time needed for finding maximum and index over processors as functions of $P$ for different degrees of the array of trees. For $P > 800$, only part of the cases were tested.

**Measured performance of the finding the maximum in the F-PRAM emulator**

Since we are finding a maximum over processors, there is no such thing as speedup. Instead we just measure how long it takes to find the needed maximum with the corresponding index. Here in the first test we had the degree $d$ as another parameter instead of assigning it a parameter-dependent value as, e.g., in Equation (7-63). For actual use in the LU decomposition we used the assignment

deg := min(P, max(2, L / max(2*B, 67))); (7-65)

to set the near optimal degree. Figure 7-23 presents the time needed as a function of $P$ for different degrees. As expected, the functions have constant steps as $\lceil \log_d P \rceil$ grows. The non-vertical steps are due to plotting with interval 1 and the logarithmic scale. By inspecting the measured data, we can rather accurately tell the constants of the execution. For example, the degree-2 plot follows the function

$$105 + 74 \times \lceil \log_2 P \rceil, \tag{7-66}$$

and the degree-8 plot follows the function

$$105 + 67 \times 8 \times \lceil \log_8 P \rceil. \tag{7-67}$$

Even if the compacted version of the algorithm does not include it, we used an additional *if* statement, which skips the *for* iteration if degree = 2. This way the constant of the algo-

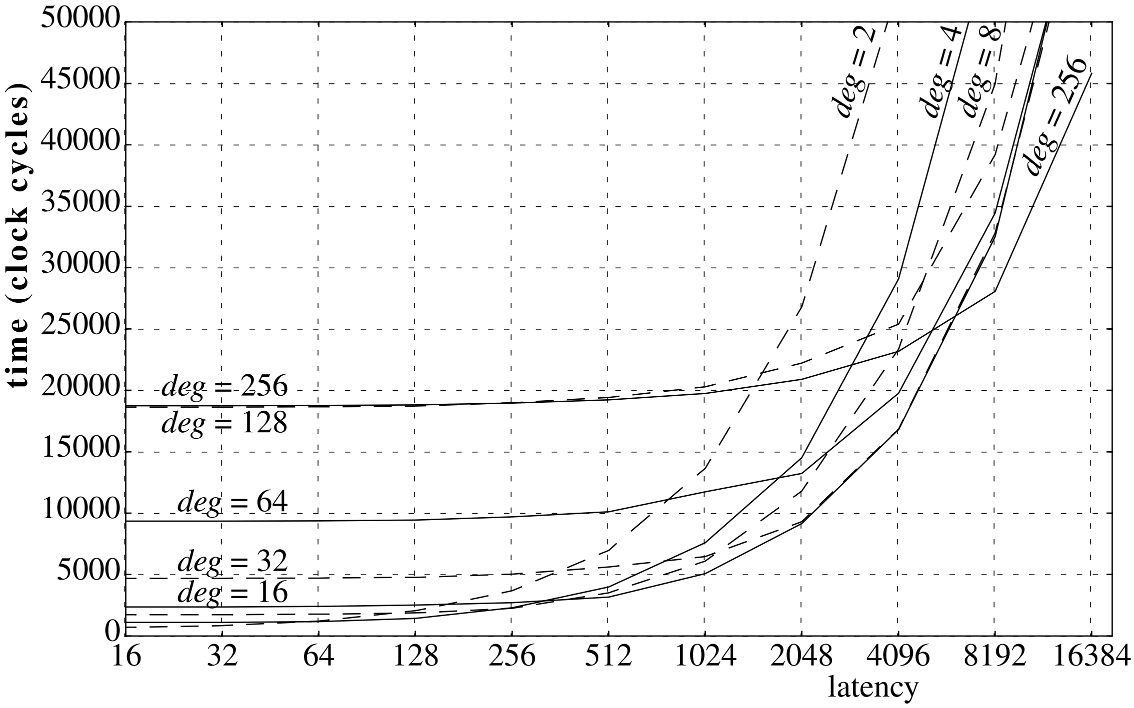


**Figure 7-24:** Time used for finding the maximum as function of latency for different degrees, $P = 256$, linear y-scale.

rithm is slightly lower for $d = 2$. Otherwise, the use of the degrees 3, 4, and 5 would have been slightly faster than the use of the degree 2 for most values of $P$.

Inspecting Figure 7-23 we could conclude that normal binary array of trees would be all we need to examine. But recalling Formula (7-58), we still have to study the impact of latency. Figure 7-24 presents the impact of latency in the finding the maximum for different degrees of the array of trees. By using $d = 256$ we went to the extreme with $d$ as high as $P$, i.e., each processor reads the all other values to find the maximum with a single latency. This approach is the fastest one if $L$ is very large. For small values of $L$ or $d$, Figure 7-24 is too inaccurate. Magnifying the lower parts of the graph, Figure 7-25 presents the same graph using logarithmic $y$-scale. Now we can see that even with not very large latencies, the degree 4 version is faster than the degree 2 version.

We could not previously give a definitive expression of the optimal value of $d$ in the algorithm. Now as we have measured the performance of the algorithm for different values of $d$, $L$, and $B$ we can conclude the best value of $d$ for each combination of $L$ and $B$. Table 7-3 presents the best values of $d$ for a set of values of $B$ and $L$. Note, however, that we tested only values of the form $d = 2^n$, which are believed to be the best ones. The table follows rather closely the predicted values of Table 7-1 with the exception that the estimated values were lower than necessary because of the simplified formula used. Table 7-4 presents the ratios of the times measured by using the degrees given using expression (7-65) to the times measured by using the best degrees. Notice that in most cases the difference is insignificant. In most cases the used degrees are the optimal ones, and the small differences (20 clock cycles) is due to the cost of evaluating the new degree with expression (7-65). For some combinations of $B$ and $L$, however, Formula (7-63) gives too low

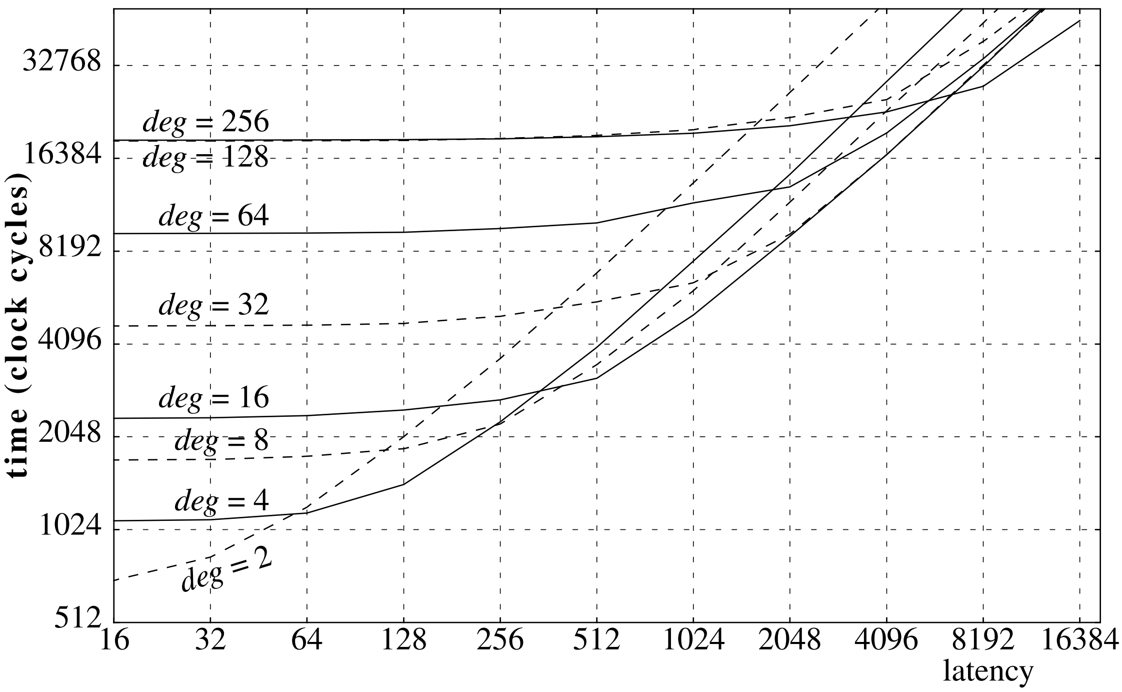


**Figure 7-25:** Time used for finding the maximum as function of latency for different degrees, $P = 256$, logarithmic scales.

**Table 7-3:** Optimal (power of 2) values of $d$ for different values of $L$ and $B$, finding the maximum, $P = 256$.

| Latency | Bandwidth inefficiency $B$ | | | | | | |
|---|---|---|---|---|---|---|---|
| $L$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| 8 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 16 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 32 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 64 | 4 | 4 | 4 | 4 | 2 | 2 | 2 |
| 128 | 4 | 4 | 4 | 4 | 4 | 2 | 2 |
| 256 | 8 | 8 | 8 | 4 | 4 | 4 | 2 |
| 512 | 16 | 16 | 16 | 8 | 4 | 4 | 4 |
| 1024 | 16 | 16 | 16 | 16 | 8 | 4 | 4 |
| 2048 | 16 | 16 | 16 | 16 | 16 | 8 | 4 |
| 4096 | 32 | 32 | 32 | 32 | 16 | 16 | 8 |
| 8192 | 256 | 256 | 256 | 256 | 16 | 16 | 16 |
| 16384 | 256 | 256 | 256 | 256 | 16 | 16 | 16 |

values. Thus, we should find a way to round up the values to conform with Formula (7-64).

**Table 7-4:** The ratios of the execution times of finding the maximum with estimated (as Formula 7-63) values of $d$ (Table 7-1) to the optimal degrees (Table 7-3) for different values of $L$ and $B$, $P = 256$.

| Latency | Bandwidth inefficiency $B$ | | | | | | |
|---|---|---|---|---|---|---|---|
| $L$ | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| 8 | 1.028 | 1.028 | 1.026 | 1.018 | 1.011 | 1.006 | 1.003 |
| 16 | 1.028 | 1.027 | 1.025 | 1.018 | 1.011 | 1.006 | 1.003 |
| 32 | 1.024 | 1.023 | 1.022 | 1.017 | 1.011 | 1.006 | 1.003 |
| 64 | 1.017 | 1.017 | 1.028 | 1.055 | 1.010 | 1.006 | 1.003 |
| 128 | 1.379 | 1.379 | 1.377 | 1.251 | 1.034 | 1.005 | 1.003 |
| 256 | 1.408 | 1.408 | 1.411 | 1.301 | 1.249 | 1.010 | 1.003 |
| 512 | 1.086 | 1.086 | 1.093 | 1.009 | 1.005 | 1.238 | 1.008 |
| 1024 | 1.391 | 1.391 | 1.426 | 1.405 | 1.003 | 1.003 | 1.244 |
| 2048 | 1.045 | 1.045 | 1.083 | 1.129 | 1.002 | 1.002 | 1.001 |
| 4096 | 1.145 | 1.145 | 1.166 | 1.263 | 1.061 | 1.001 | 1.001 |
| 8192 | 1.455 | 1.444 | 1.457 | 1.345 | 1.202 | 1.128 | 1.001 |
| 16384 | 1.874 | 1.874 | 1.840 | 1.820 | 1.285 | 1.176 | 1.100 |

## 7.5 Software synchronization

Every parallel program that uses shared memory needs synchronization to ensure that the writes to and reads from the shared memory occur in correct order. The F-PRAM model requires that each F-PRAM machine has a synchronization facility that can synchronize all processors in time $S$. Some parallel machines, however, do not have a dedicated synchronization facility. Therefore we have to implemented our own synchronization routine to make sure that we can port the F-PRAM model to such parallel machines. Furthermore, we can use our own algorithm for random submachine synchronization, which is not possible with the native synchronization facilities of typical parallel computers.

The synchronization of the processors consists of two stages. Firstly, we have to ensure that all processors have started the synchronization. Secondly, the knowledge of the completion of the first phase must be spread to all processors. This can be done using a shared vector which has an element for each processor. The elements are initially zeroes. As soon as a processor enters the synchronization procedure, it writes a 1 to its own synchronization vector element. The detection of the arrival of the signals of the processors can be done in a binary tree fashion. After one processor has ensured that all processor signals have arrived, the knowledge is again spread using another binary tree. As with the finding the maximum, also these synchronization trees can be combined to one $d$-ary array of trees to reduce the impact of the latency. As a matter of fact, the synchronization algorithm resembles the algorithm to find the maximum a lot. The differences are only the operations within the innermost loops and the additional *repeat-until* waiting for the possibly missing predecessor.[73]

```
procedure softsync();
    fwrite all syncarray[PID(0)] := 1;
    sum := 1;
    if deg > 2 then          // a vector version for deg > 2 (deg-ary trees)
        while sum < P(0) do  // repeat while not all processors have signalled
            repeat                 // repeat until all predecessors have signalled
                for i := 1 to deg-1 do
                    future all T1[i] := syncarray[(sum * i + PID(0)) mod P(0)];
                fail := false;
                for i := 1 to deg-1 do
                    if T1[i] < sum then
                        fail := true;
            until not fail;
            sum := sum * deg;
            fwrite all syncarray[PID(0)] := sum;
    else          // simpler version for deg = 2 (binary trees)
        while sum < P(0) do  // repeat while not all processors have signalled
            repeat                 // repeat until the predecessor has signalled
                future all t1 := syncarray[(PID(0) + sum) mod P(0)];
            until t1 >= sum;
            sum := sum * deg;
            fwrite all syncarray[PID(0)] := sum;
```

**Algorithm 7-9:** Software synchronization procedure to be executed by every processor.

Algorithm 7-9 presents the code for the synchronization algorithm. This version neither includes the initialization, nor the clearing of the array used for synchronization detection. The initialization has to be done at the beginning of the whole program, i.e., in the main program.[74] The clearing of the vector after each synchronization is more difficult. Simply assigning zero at the exit of the procedure will not do since some processors may still need the value. The clearing at entry is even worse. Consequently the clearing has to be done outside this low-latency procedure. The solution is to wrap this routine with a procedure that recycles three different synchronization arrays and presents a clean one for the main synchronization procedure each time it is called. Two arrays is not enough if we have long latencies with very large variations, a change in the order of references, and two synchronizations very close to each other.

The use of the software synchronization is similar to the default synchronization primitive of the F-PRAM model, i.e., all processors need to enter the procedure, otherwise the procedure deadlocks. The difference is that the software version does not officially guarantee the finishing of all shared memory references. In practice, however, the proce-

73. Finding the maximum used the synchronization primitive to ensure the correct order of references.

74. Or called separately from the main program.

dure takes at least $(C+2\log P)\times L$ time to complete and it makes $O(P\log P)$ references itself, and, thus, the earlier references should be completed if we can assume any fairness. The advantage of the software version is that it can be used for submachine synchronization easily by using $P$ and $PID$ instead of $P(0)$ and $PID(0)$. To concurrently synchronize several submachines, the use of the shared vector should also be revised.

If the software synchronization algorithm resembles the algorithm of finding the maximum a lot, the analysis of the two algorithms is even more similar. Especially the analysis of the proper values of the degree of the array of trees ($d$) is the same, and, thus, we shall not repeat it here. We only need to state that the synchronization algorithm takes time

$$S_S = \max(L, C_s\times d)\times(\lceil \log_d P\rceil + 2), \tag{7-68}$$

where $C_s$ is a local constant. For analysis purposes we can make an inaccurate assignment $d = O(L)$, after which we can state the asymptotic time complexity of

$$S_S = O(L)\times O(\log_L P) = O\left(\frac{L\log P}{\log L}\right), \tag{7-69}$$

which, however, is of little use because of its inaccuracy. The additional wrapper procedure to coordinate the resetting of the arrays uses only $O(1)$ time for each processor.

In addition to synchronization and finding the maximum, the same algorithm skeleton can be applied to, e.g., summing and broadcasting between the processors. Vectors larger than $P$ have to be first locally summed, and then summed over the processors.

### Measured performance of the software synchronization in the F-PRAM emulator

The definition of the delay of the synchronization is the time needed after the last processor starts to execute the procedure. After that, the procedure makes $\lceil \log_d P\rceil$ iterations of the main *while* loop. Consequently, neither the order of the arrivals of the processors, nor the variation of the arrival times impact the number of the main iterations. On the other hand, the waiting iterations within each main iteration may occur randomly one or more times depending on the arrival times. Especially, the times are multiples of the inner iteration times, which are quite large if $d$ is large. Consequently, the time required for the synchronization varies a bit from run to run depending on the arrival times.

In the following measurements, the time is taken for each processor before and after the procedure call. The time used in the graphs is the difference of the last arrival time and the last exit time. Before the synchronization call each processor was made to do random work. Thus, the processors entered the procedure in random order. Figure 7-26 presents the basic performance of the procedure for different values of $d$ and $P$. The bounds of the random variation are clearly visible. The few spikes above the standard levels are probably due to unexpectedly long latencies or temporary network saturation. Within the most interesting domain $P = 16$-$2048$, we can conclude that $S_S = 256$-$512$, if the latency is low. Figure 7-27 presents the impact of latency for the synchronization for a fixed $P$. The results are very similar to those of the finding the maximum, i.e., for $P = 256$, the useful values for degree are 2, 4, 16, and 256, depending on the latency.

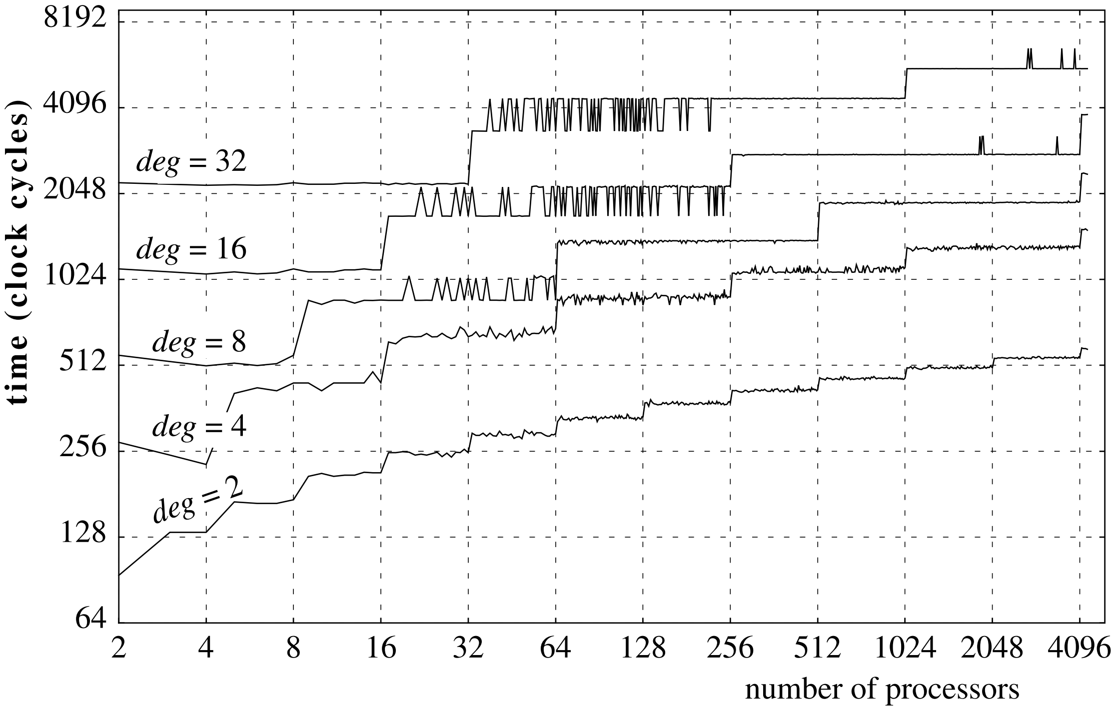


**Figure 7-26:** Time needed by software synchronization as a function of the number of processors for different degrees.

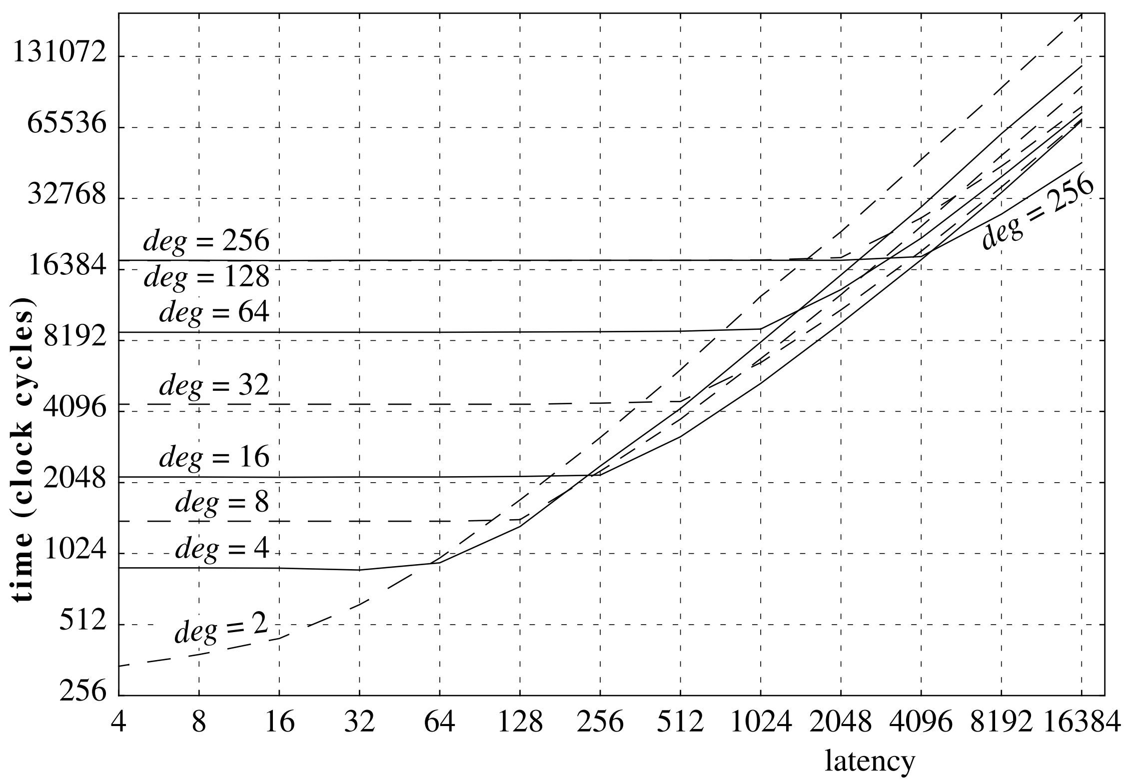


**Figure 7-27:** Time needed by software synchronization as a function of latency for different degrees, $P = 256$.

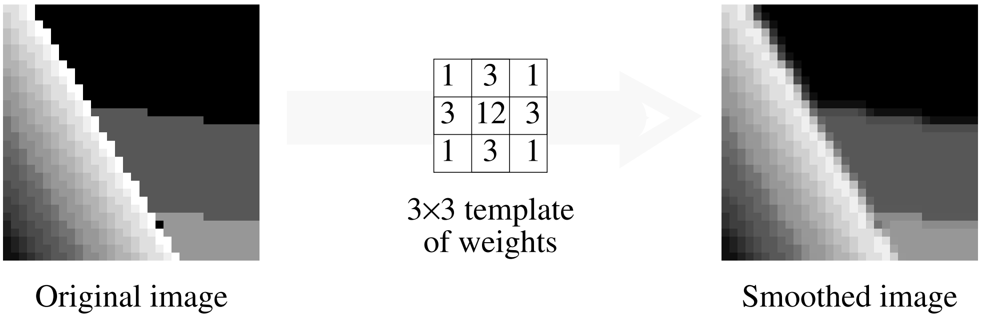


**Figure 7-28:** Image smoothing example with 32×32 pixel original image (20 levels of gray), 3×3 template of weights and the smoothed image.

As we saw with the analysis of the finding the maximum, the evaluation of the correct degree of spreading is not a trivial task. Especially for a simple atomic frequent operation such as synchronization it seems to be a bit too difficult. In practice, however, the optimal $d$ has to be decided for each machine just once, and used thereafter. We can test beforehand all candidates from 1 to $P$, and use the best within our synchronization routine. Especially if $P$ is not a power of two, we can use a degree that is not a power of two. For example, if $P = 24$, the synchronization is faster with $d = 5$ than with $d = 4$. Moreover, if $L > 50$ it is also faster than with $d = 2$.

## 7.6 Image smoothing

Many signal processing problems are often characterized as easily parallelizable problems. As an example of a digital signal processing problem we present an image smoothing algorithm. The input is an $N \times N$ pixel grayscale image presented as one integer for each pixel. In the result image each pixel is computed as a weighted average of the corresponding original pixel and the surrounding pixels of the original image. The weights are given in a template of size $M \times M$, where $M$ is odd, and the original pixel is matched with the center weight of the template. Typical values for $M$ are 3, 5, and 7. These types of filters are typically used for reducing transmission and digitalization noises. Figure 7-28 presents a magnified fraction of a grayscale image, a smoothing template of size 3×3, and the corresponding smoothed image fraction. The smoothed image of the example remains still quite pixelized because of the magnification (low resolution) and the low number of distinct grayscales. Note, that this smoothing does not increase resolution or color depth. However, the same algorithm can be used to improve the apparent image quality by dividing each pixel into four pixels, and/or doubling the number of grayscales of the original image before smoothing.

### F-PRAM implementation

Since the computation of each pixel depends only on the values of the $M \times M$ neighboring pixels in the original image, the smoothing can be easily parallelized. We divide the image to $\lfloor \sqrt{P} \rfloor \times \lfloor \sqrt{P} \rfloor$ square blocks of size

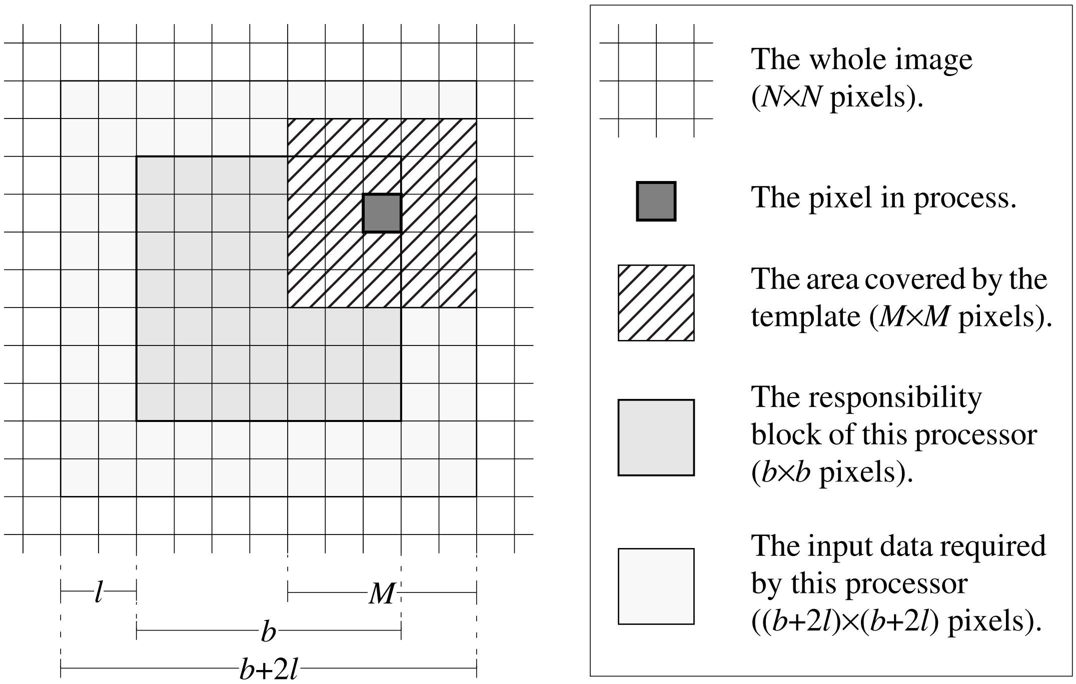


**Figure 7-29:** Examples of the variables in the image smoothing, 5×5 template, 7×7 pixel block for each processor.

$$b \times b = \left\lceil \frac{N}{\lfloor \sqrt{P} \rfloor} \right\rceil \times \left\lceil \frac{N}{\lfloor \sqrt{P} \rfloor} \right\rceil \tag{7-70}$$

each. For simplicity we shall assume that $N = b \times \sqrt{P}$, i.e., the image can be divided to blocks of equal size. The division of the result image among the processors equal the division of the matrix multiplication in Section 7.2. The difference is the amount of input data each processor needs. In matrix multiplication each processor needed slices of both input matrices (see Figure 7-8) totalling input data volume of $2\sqrt{P} \times b^2$. In image smoothing each processor needs only its own block and a few neighboring values immediately outside the block. The outside pixels are needed for the average calculation of the outer pixels of the block. More accurately, for template of size $M \times M$, each processor needs $l = (M-1)/2$ rows and columns around its own block. Figure 7-29 represents a diagram of the data requirements for each processor. The difficulties arise for those processor that have their block on the edge of the image. On the edges of the whole image the templates need to be cut according to the image edges to avoid corruption of the outer pixels. Therefore, the processors need to check in all stages whether there are neighboring pixels or not. Unless $P$ is very small (e.g., 4), the impact of the outside borders is negligible on the analysis as there are always processors having their block in the middle of the image. Moreover, handling the exceptions on the outside borders take some time, which balances the differences.

The practical implementation of the smoothing algorithm is very straightforward with the exception of the handling of the outside edges. In fact half of the actual code is *if-then-else* sequences to handle these exceptions. Algorithm 7-10 presents the body of the program without the most complex exceptions. Especially the fetching of the neighboring

```
program smooth;

var own : array[0..b+M-2, 0..b+M-2] of word;
    newown : array[0..b-1, 0..b-1] of word;
    template : array[-l..l, -l..l] of word;
sharedvar borders : array[0..N-1, 0..N-1] of word;

    input template;

    read all(x);      read all(y);      // left-upper corner point of the block

    for i := 0 to b-1 do                // input own block of image
        for j := 0 to b-1 do
            read(own[i+l, j+l]);
            if within_the_inside_borders then
                fwrite all borders[x+i, y+j] := own[i+l, j+l];

    synchronize;

    fetch neighbouring pixels from borders to the edges of own
        up, left, right, bottom; l*(b+l) pixels each (if needed)

    temsup := sum(template);
    for i := 0 to b-1 do            // for each
        for j := 0 to b-1 do        // pixel of the block

            localtemsum := temsum;   sum := 0;
            for ii := -l to l do            // for the
                for jj := -l to l do        // whole template

                    if within_the_image then
                        inc(sum, own[i+ii, j+jj]*template[ii, jj]);
                    else
                        dec(localtemsum, template[ii, jj]);

            newown[i-l, j-l] := round(real(sum) / localtemsum);

    output newown;
```

**Algorithm 7-10:** Image smoothing algorithm, $N$, $M$, $b$, $l$ as in the text.

pixels would add another page of code if displayed fully here. Additionally, in our full version, if $P = 1$, the processor skips all the parts concerning the borders, which improves the efficiency a bit. Even if the shared array *borders* covers the whole image, the processors assign to it only the values that are needed by other processors. The innards of each block are input locally, used only locally, and output locally, i.e., they never reach the shared memory.

Note that this presented algorithm does not need any *par-do* statements as the input is already divided to $P$ blocks and each processor reads all needed values with the input. As there are no *par-do* statements, all processors execute the same code. We could have included two nested *par-do* statements for $x$ and $y$, but the program would not have used them, i.e., it can do without them as well.

**Analysis**

The work (and sequential time complexity) of the smoothing is $N^2M^2$ steps for image of size $N \times N$ and smoothing template of size $M \times M$.[75] In Algorithm 7-10 the actual smoothing (lines 17-26) is trivial to parallelize optimally. Thus, the smoothing executes in

$$T_{sm} = \frac{N^2M^2}{P}, \tag{7-71}$$

steps as long as $P \le N^2$. Similarly the input and outputs, which sequentially take $N^2$ steps each, parallelize fully, and we shall amortize these to the smoothing cost (7-71). Using sequential I/O would destroy the efficiency if $P > M^2$. Moreover, the distribution of the data via the shared memory would take much more time and require more bandwidth.

The main cost of the parallelization is the need to exchange the borders before the actual smoothing. The writing of the borders to the shared memory is done during the input and it does not take additional iterations or much additional time. The fetching of the borders is a bit more difficult. If the block of a processor is not on the edge of the image, the processor needs to fetch $l = (M\text{-}1)/2$ rows and columns around its own block as seen in Figure 7-29. The number of border pixels to fetch, and, thus, the number of steps required, is

$$T_{bofe} = 4 \times l \times (b+l) < \begin{cases} 2 \times b \times M = 2 \times \frac{N}{\sqrt{P}} \times M & \text{if } b \ge \frac{M}{2}\text{, i.e., } P \le 4\frac{N^2}{M^2} \\ 2 \times M^2 & \text{otherwise.} \end{cases} \tag{7-72}$$

The latter case is rare because it assumes a very large number of processors. Thus, the whole program takes

$$T = T_{sm} + T_{bofe} = \frac{N^2M^2}{P} + 2 \times \frac{N}{\sqrt{P}} \times M = O\left(\frac{N^2M^2}{P}\right) \quad \text{if } P \le N^2, \tag{7-73}$$

i.e., it is asymptotically work optimal. The number of steps in the block fetching stage also determines the number of shared memory operations, i.e., the bandwidth needed. Therefore, we can rewrite the execution time (7-73) to form

$$T = C_{sm} \times \frac{N^2M^2}{P} + \max(L, \max(C_{bf}, B) \times 2 \times \frac{N}{\sqrt{P}} \times M), \tag{7-74}$$

if we want to take the F-PRAM parameters into account. The practical requirements on $B$ and especially on $L$ are very loose unless $P$ is close to $N^2$.

75. If we used uniform (nonweighted) template, the asymptotic work could be reduced by at least a factor of $M$.

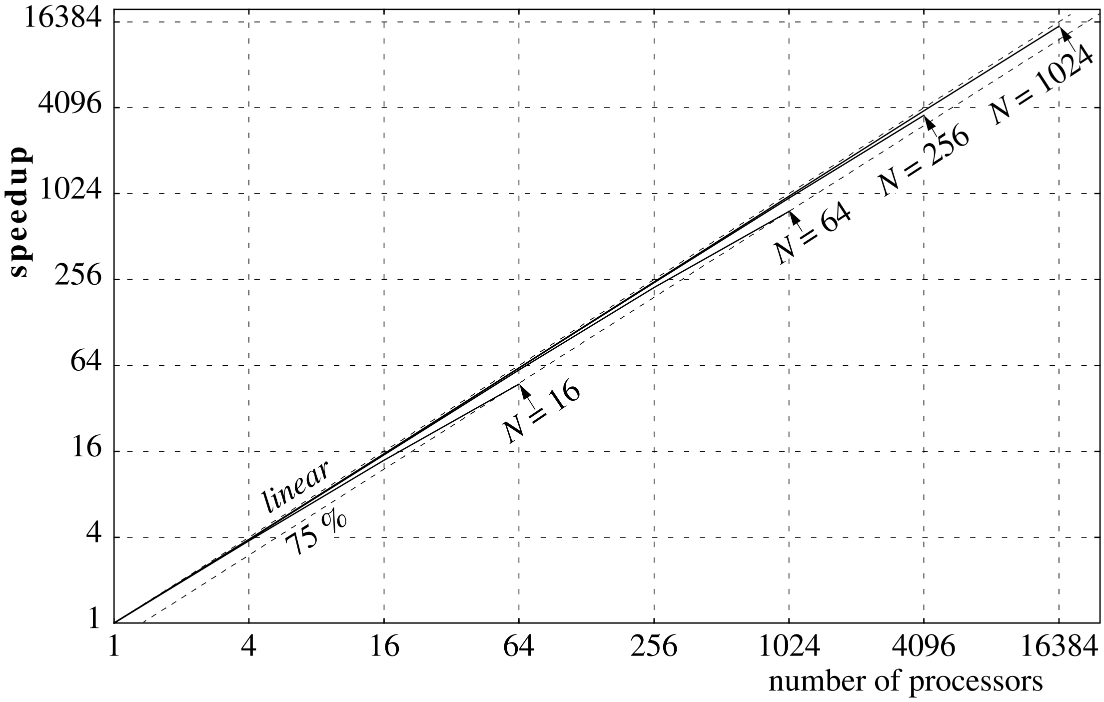


**Figure 7-30:** Speedup of the image smoothing as a function of the number of processors for different input sizes, $M = 5$.

### Measured performance on the emulator system

The smoothing program executes deterministically, independently of the input data. For the measurements we used random images. As the above analysis suggests, the image smoothing is an embarrassingly parallel problem. We shall not give it the full treatment as we gave to the other problems. Instead, we shall only present the vast potential parallelism of the image smoothing. Figure 7-30 presents the basic speedup of the algorithms for $M = 5$. The algorithm parallelizes so well that is a bit difficult to plot properly. It suffices to state that the efficiency remains above 75 % unless $P = N^2$. At the extreme, for $N = 1024$, $P = 16348$ (8×8 blocks), the speedup was 15290, translating to 93 % efficiency.

The impact of the template size $M$ to parallelization is quite small. Even if the amount of border data increases linearly[76] with $M$, the work of smoothing increases quadratically with $M$. The asymptotic difference does not show up severely since $M \leq 7$ in practice. The measurement data show that the speedup for $M = 7$ is the best one, for $M = 5$ the speedups are 2-3 % lower, and for $M = 3$, 7-12 % lower. Plotting these curves for this large range of $P$ is useless since the differences are barely visible on the logarithmic scale required for large range up to $2^{14}$ processors.

The impact of $L$ to the image smoothing is similar to the matrix multiplication, i.e., no significant effect unless the single latency dominates the execution time. Also the impact of $B$ is not very severe unless $P \geq N^2/M^2$, i.e., the whole image needs to be transferred at least once. Figure 7-31 presents the impact of $B$ for different values of $P$. Even with $N = 256$, $P = 1024$ (8×8 blocks), only bandwidth inefficiencies beyond 1000 impact

76. Quadratically, if $P = N^2$

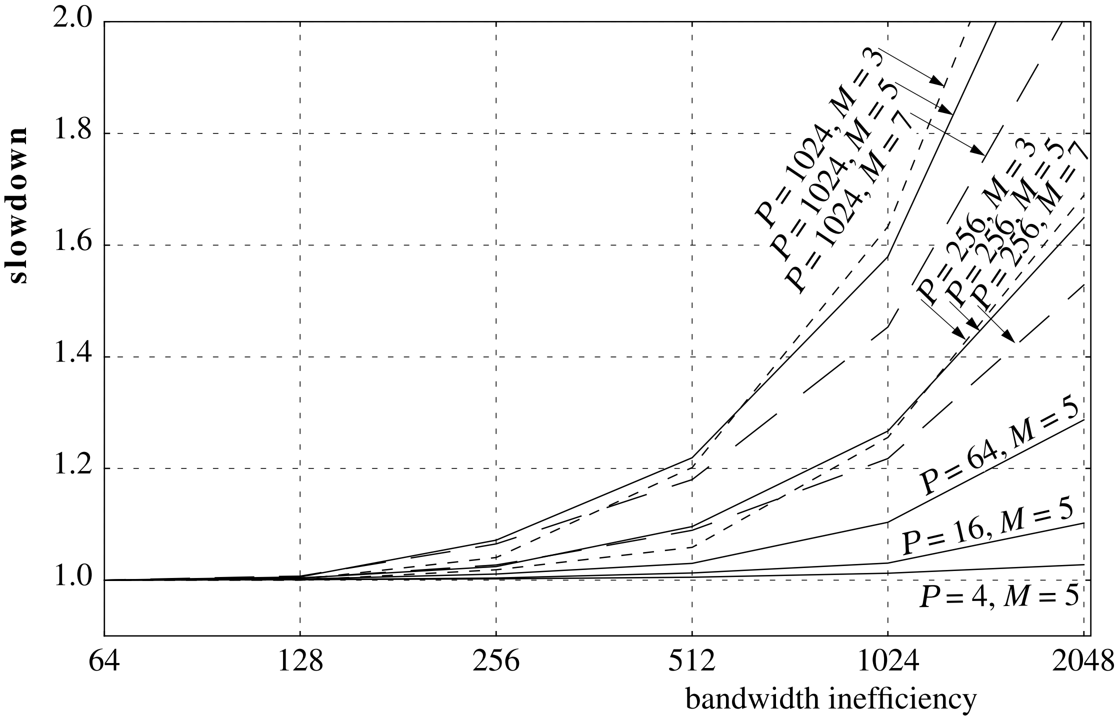


**Figure 7-31:** Slowdown of the image smoothing as a function of bandwidth inefficiency for different numbers of processors and template sizes, $N = 256$, magnified y-axis.

the performance significantly. As suggested by Formula (7-74), even if the increase on template size increases total communication volume, it increases total work even more, unless $P \geq N^2/M^2$. Figure 7-31 presents also the impact of $M$ for $P = 1024$ and $P = 256$. Note that the y-scale of the graph is very magnified compared to the earlier graphs. The differences on the curves are actually very small. For $P \leq 64$ the lines for different values of $M$ were too close each other to be clearly plottable even on this scale.

# Chapter 8

# Modeling the existing parallel computers with the F-PRAM model

We introduced the F-PRAM to model both parallel computations and parallel computers. In this chapter we shall use the F-PRAM to model some of the existing parallel computers. More accurately, we shall present some estimations on the values of the F-PRAM parameters for some of the existing parallel computers. These estimations are not very accurate, but it is interesting to compare these values to the graphs of the Chapter 7.

Even if the classification is not very clear, or even unambiguous, we shall divide the parallel computers in four classes based on the implementation of the shared memory. In Sections 8.1, 8.2, 8.3, and 8.4 we shall discuss shared memory parallel computers, virtual shared memory computers, distributed memory computers, and networks of workstations, respectively. As we have noted earlier, the F-PRAM model does not dictate the method of the implementation of the shared memory. In fact, each of the parallel computers to be presented in this chapter has a more or less unique solution for the interprocessor communication. Furthermore, we discussed the efficient simulation of the F-PRAM with message passing system in Subsection 4.7.3. Consequently, we shall express the communication in F-PRAM terms, no matter whether the machines have shared memory, virtual shared memory, or distributed memory.

The only accurate way to estimate the values of the parameters of the F-PRAM model for real parallel computers would be to implement the model primitives somehow and measure the values. Moreover, the implementation of the model should be close to optimal to provide valid information on the machine. Since so extensive measurements were not possible within this work, we have to restrict ourselves to estimations mostly based on the information provided by the parallel computer manufactures. The marketing information naturally does not provide very accurate information for all F-PRAM parameters. Especially the scalability of the information is questionable. Consequently, we shall only make rough estimations on the most important parameters based on the available information. In most cases we use the maximum peak performance of the hardware. Even if the maximum speed is unattainable in real software, it is less dependent on the measuring method than different end user throughputs and latencies. Because the estimations are based on different sources, we avoid accurate comparisons of the different computers. Especially, we did not want to put the values of the different computers side by side to a single table, even if it would make the comparison easier. Examples of such tables can be seen in [27, 48]. Hopefully we can make more reliable estimations on the subject in future research.

In Section 8.5 we shall sketch a shortcut implementation of the F-PRAM model for the Cray T3E computer. This version would have been used for real world measurements if the T3E libraries had supported asynchronous gets from the memory of another processing node.

## 8.1 Shared memory computers

In this section we shall present computers that have a separate shared memory, to which the processors are connected. The shared memory can be implemented either as a single memory bank, or more scalably as a set of memory banks operating in parallel. The connections between the processors and the memories can also be implemented using different types of networks. The simplest one is a bus to which the processors are connected. More scalable computers usually exploit different types of multistage networks.

### Vector supercomputers

As an example of the traditional vector supercomputers, we use the Cray Y-MP T90 series. This system can have up to 32 processors, which are connected via a multistage network to the shared memory consisting of several memory banks. On each cycle every processor is able to do two arithmetic operations with the vector unit and a memory connection through all of the four memory channels. Consequently, we can estimate that $B \approx 0.5$. Normally the latency is hidden in vector computer by pipelining the vector operations. Because this approach can also be used when simulating the F-PRAM model, we get $L \leq 64$.

### Tera MTA

Tera computer system is a very ambitious project to produce parallel computers based on multithreading processors [8]. Because currently (early 1998) only one working processor has been installed, the actual parallel machines have not yet been built. Hence, we can discuss only the projected performance. The system uses a sparse 3D-torus network with separate nodes for the memory modules and the processing nodes. The number $p$ of processing nodes will vary between 16 and 256. The communication network is a sparse 3-dimensional torus containing $p^{3/2}$ nodes. The network is not a full 3D mesh, since alternating $x$ and $y$ connections are omitted depending on the parity of the $z$ connection. In other words, each of the nodes has been divided to two in the $z$ dimension. In addition to the processing nodes, the network includes $2p$ or $4p$ memory nodes and $2p$ I/O nodes. The processors and the network nodes operate at 3 ns clock cycle with 64-bit words. Each of the processors includes 128 contexts, or threads,[77] and is able to change its context after every instruction without any additional delays. We shall first consider the system without

77. Thus the name Multi Threaded Architecture, MTA.

the multithreading operation. Each of the $p^{3/2}$ nodes is able to deliver one word with addresses in all four directions in every cycle. Since the average bi-directional distance for each packet is about $2p^{1/2}$, the interconnection system has capacity of at most $4p^{3/2}/2p^{1/2} = 2p$ new words per clock cycle, i.e., 2 words per processing node per cycle. Since the processors are able to issue one reference per clock cycle, we get the excellent value $B = 1$. The worst case latency of a bi-directional packet is $9p^{1/2}$ cycles, which translates to 144 cycles on a 256 processor machine without taking the memory latency into account. Since the system uses DRAMs, we can estimate that in total $L \approx 160$. If we exploit the multithreading facility of the MTA architecture, the latency as seen by one thread drops to 2. Machines with fewer than 256 processors have latency one for the 128 threads. The algorithm-dependent question is whether the latency or the number of processor is easier to handle without loss of efficiency. For example, we can use either a machine with $P = 256$, $L = 160$, or a machine with $P = 32768$, $L = 2$ having 128 times slower processors. The data of our experiments suggests that the large increase in $P$ often impacts more on efficiency than the not-too-large $L = 160$.

**Silicon Graphics Power Challenge**

As an example of the classic bus-based symmetric multiprocessor (SMP) systems we introduce the Silicon Graphics Power Challenge system [94]. The system has up to 18 processors of type MIPS R8000 connected to a shared memory via a common bus. The processors are superscalar and have up to 300 MFLOPS power at 75 MHz clock speed. Here we use a more modest 150 MFLOPS processing power. The memory reference latency to the shared memory, i.e., not to the caches, is 53 cycles. With the two instructions per clock cycle, this translates to $L = 106$. The system bus and the memory are able to serve up to 9.5 million transactions, i.e., cache fills, per second, which translates to $B \approx 284$. Each of the cache fills consists of 128 bytes, which is unnecessarily large for the basic 16-byte future references. By using the block transfers we can improve the performance considerably. Since the 128 bytes are transferred in a single block, we achieve $B_B \approx 36$ for packets having proper sizes. Generally, the multilevel cache subsystems of all SMP computers complicate the estimation of the shared memory performance. Especially the marketing information given by the manufacturers relies heavily on the cache usage. The memory transaction rate should be, however, a quite good and reliable measure, and we feel that it could be a useful general measure of the memory performance.

**Sun HPC 10000**

As an example on how the architectures of the workstation-based servers and the more traditional supercomputers converge to the same direction, we present the Sun HPC 10000 (aka the Starfire) high-performance computing server [97]. The computer can be configured to have up to 64 333 MHz Ultrasparc processors. As earlier, we use a more modest 300 MFLOPS power instead of the advertised 500 MFLOPS peak. Instead of using a single very fast bus between the processors and the memory, the processors are connected with a crossbar switch. The switch provides 200 MB/s bandwidth for each processor, which stands for $B = 24$, which provides much better communication facilities than the bus-based approach for this large $P$.

## 8.2 Parallel virtual shared memory computers

In this class the memory is distributed to the processing nodes, but the messages induced by the shared memory references are handled with dedicated hardware. Most importantly, the hardware of the interconnection network is optimized for messages induced by the shared memory references, i.e., it can handle small packets and has a reasonably low latency.

### Thinking Machines CM-5

The CM-5 parallel computer manufactured by Thinking Machines Corporation [98] was one of the first massively parallel MIMD computers. The architecture supports up to 16,384 nodes,[78] but the largest one built has only 1,024 nodes. The nodes of the CM-5 are connected by a fat-tree interconnection network, i.e., a tree that has several parallel links and nodes on higher levels of the tree. The processing nodes are located on the leafs of the tree. Since the number of upper branches of the fat tree grows with the number of nodes below, the bandwidth remains constant 5 MB/s (≈ 0.6 MW/s) per processor. Each of the processing nodes consists of a RISC processor for scalar operations, up to four vector units and up to four memory banks. Each of the nodes has about 128 MFLOPS (or 128 MOPS) of peak processing power, and, thus, we get $B \approx 200$. Besides the main interconnection network, the CM-5 also has a control network for barrier synchronization and a network for parallel prefix (and suffix) operations on simple operations. Because of the dedicated synchronization network, we can state that $S$ is a small constant. According to [27], the unidirectional hardware latency between the processors can be as low as 246. Consequently, we can approximate $L \approx 500$ for the shared memory access.

### Cray T3E

The T3E parallel computer [25] manufactured by Cray Research is currently one of the most sold MPP computers available. The T3E can contain up to 2,048 nodes, each of which includes a Digital Alpha (21164 or 21164A) microprocessor. The processor speeds range from 300 MHz to 600 MHz depending on the model. The newest one is the T3E-1200 having theoretical 1200 MFLOPS peak processing power. A more realistic estimation of the speed is 600 MFLOPS. The nodes are connected with a 3-dimensional torus network with each of the links having 480 MB/s maximum bandwidth to both directions. The communication subsystem is able to route references to the other nodes autonomously, resulting in 2.88 GB/s = 180 Mpackets/s per node. Here we shall consider a 1,024-processor T3E. The diameter of the network is 16, and an average packet goes through 16 links on a two-way route. Consequently, the system can route at most 180/16 = 11.25 million global packets per second per node. Using the processing power 600 MFLOPS, we get a lower limit $B = 54$ for the bandwidth inefficiency. The latency of a shared memory read using the SHMEM library is reported to be $L = 450$ for the older generation (375 MHz version) [26].

78. In theory, the architecture supports up to 262,144 nodes, but the wire lengths of the network would be too long.

### Fujitsu VPP700E

The Fujitsu VPP700E [35] is a hybrid of the vector supercomputers and the MPP computers. Each processing node of the VPP700E is a vector supercomputer with 2.2 GFLOPS of processing power. The machine can have up to 256 processing nodes connected through a crossbar network. The network has 615 MB/s bandwidth for each processor. Using 16 bytes/packet, we get $B = 57$, which is close to the Cray T3E value, but is more scalable and better guaranteed because of the crossbar topology. Also the latency should be much less because of the unit diameter.

## 8.3 Distributed memory parallel computers

As the last class of the actual parallel computers, we shall present some message passing computers. In this class of computers the processors communicate by exchanging messages instead of being able to reference the memory located in other processing nodes. In some cases the processing nodes participate in the routing of the packets destined to other nodes. Typically, the best performance on communication between the nodes can be achieved using rather long messages.

### IBM SP/2

As an example of a classic[79] message passing parallel computer we look at the IBM 9071 SP/2 parallel computer [2]. The nodes of the SP/2 are based on the IBM RS/6000 workstations. The wide nodes consist of a 66 MHz Power2 RISC processor, an amount of local memory and one to four local hard discs. The maximum power of the processor is 133 MOPS or 266 MFLOPS. The interconnection network is a multistage omega network made of $8 \times 8$ crossbar switches. The maximum per processor bandwidth of the network is 40 MB/s and the reported application-to-application bandwidth is 10 MB/s. The application-to-application latency is reported to be about 40 μs for a full configuration. Using the more modest 133 MOPS performance estimation and bi-directional communication, we get $B \approx 400$, and $L \approx 10{,}000$. We must remember, however, that the above performance figures are for application-to-application performance measured probably for the PVM system as opposed to the figures of the other machines, which are the hardware performance figures.

### Intel ASCI Red

Intel ASCI Red is a unique parallel computer ordered by the US Department of Energy for the Sandia National Laboratories, e.g., for simulating the storage of nuclear weapons [90, 74]. The computer has about 9,000 Pentium Pro microprocessors connected by a two dimensional mesh similar to the Intel Paragon parallel computer [55] it is based on. The computer consists of 4,536 processing nodes, each with 2 pieces of 200 MHz processors

79. The current versions (RS/6000 SP models) are about two to three times as fast as the earlier models.

totalling 400 MFLOPS of processing power per node. The processing nodes are connected to a three-dimensional mesh having 400 MB/s capacity per link per direction. The total bisection bandwidth of the machine is 25.8 GB/s per direction. With the previous assumptions we get

$$25.8\times10^9 \text{ GB/s} / 16\text{B/pckt} / 9072 \text{ nodes} = 0.178 \text{ Mpackets/s}, \tag{8-1}$$

which translates to $B \approx 2{,}250$ even if the network would reach its at peak performance delivering short messages. The announced topology of the network is an 38×32×2 mesh. The multiplication results in 2432, and, thus, stands for 4 processors (or 2 processing nodes) for each network node.[80] Therefore the diameter of this network is at least 70, and the total latency $L$ something of the form $c_1 + c_2 \times 140$, i.e., hundreds in practice. Especially the bandwidth does not promise good performance if we compare the value 2,250 to the values of the measurements in Chapter 7. This is because the mesh network does not have good enough bandwidth to support global communication. Instead, these types of computers have to be programmed by embedding the point-to-point communication pattern to the topology of the network. Moreover, the ASCI Red is a special purpose computer build according to the requirements of the specific problem.

## 8.4 Networks of workstations

In this section we shall consider using networks of workstations (NOW) for parallel computing and especially modeling such computing with the F-PRAM model. The workstations are designed to be used on the console by a single user, and, thus, have good user interface capabilities, especially graphics. The development of powerful microprocessors and the use of workstations for 3D-visualization and other power-demanding tasks have lead to a situation, where the different workstation manufacturers compete with the floating point processing power of their top-of-the-line workstations. Since the processing power of a single-user computer is rarely in full use even at daytime, let alone at night, the workstations form a good unused resource of processing capabilities. By using a large cluster of workstations in parallel to solve a problem, we can get supercomputer-class theoretical performance essentially free.[81] Recently, the workstation networks have been used in several very large-scale number theoretical problems. Thousands of volunteers over the Internet have provided the computing power of their workstations for distributed cryptoanalysis. The reason for the success of these attempts has been the publicity and awards for the breaking of the well known cryptosystems. These examples show that some problems can be efficiently parallelized even with only few kbits of bandwidth and latency of several seconds.

80. Some network nodes are occupied by the I/O and service nodes.

81. Naturally, the workstations are not free, but the unused power of the existing workstations is.

### Workstations

Typical current workstations have 50 to 500 MFLOPS of processing power, 64-512 MBs of memory, and several MB/s I/O speed. These features, and the graphical display, are required for their use as stand-alone workstations. When used as nodes of a parallel computer, their weak points are the communication capabilities. Typical workstation networks are connected with the 10 Mb/s Ethernet networks. The speed is good enough for loading programs from a remote disk or for remotely using another computer. More advanced networks may use a faster 100 Mb/s or 1 Gb/s version of Ethernet, a 100 Mb/s FDDI, or an ATM network. These speeds are good enough also for transferring accurate still pictures or even motion pictures in real time if the intermediate routers or collisions do not slow down the network. The networks are not, however, very good compared to the real parallel computers. Thus, we have to carefully choose the applications and the algorithms to be executed in the NOWs.

### Networks

The ordinary bus-structured Ethernet connection provides at most 10 Mb/s bandwidth between two computers. For two word packets of net length of 128 bits and headers of 64 bits, this would translate to 40,000 packets/s. In practice, however, each of the packets reserves the channel for a longer time, and the real bandwidth would be less than 1,000 packets/s. In F-PRAM terms, this means $B = P \times W/1000$, where $W$ is the processing power of the workstation in operations/second. For example, using $P = 20$ and $W = 100$ MOPS, we get $B = 2{,}000{,}000$, which means that for every communication we have to perform two million local operations to maintain full efficiency. The faster Ethernets and FDDI networks are at most one order better, which is not very good either. Especially, the number of short messages in a second is not much larger. The fundamental problem of these networks is that since they are bus-based, only one node can communicate at a time. Furthermore, the high initialization cost and the rather high minimum packet size of the FDDI network reduce the throughput using small packets. Via using a star connection and an Ethernet switch, the computers can communicate concurrently as long as the connections are point-to-point and distinct. In theory, the 20 workstations could exploit 10 connections simultaneously. In practice, however, the collisions occur, and we could hope for 5-fold increase in total bandwidth in the $P = 20$ system.

The Asynchronous Transfer Mode (ATM) network is not bus-based, but switch connected [15]. The switches on the lowest level connect the workstations together and to the upper level switches. The switches ensure uniform bandwidth for each workstation by using usually crossbar switches, which, however, restricts typical maximum switch-sizes to a dozen or two. Because of the crossbar, each of the nodes can communicate concurrently. Furthermore, the ATM network is optimized to deliver rather small packets with rather short delay. Each ATM packet is of fixed size and consists of 48 bytes of data and 5 bytes of headers, which is only about four times bigger than a standard F-PRAM packet. Consequently, the switched ATM networks fit much better for implementing the shared memory F-PRAM model than the bus-based networks. The speed and latency of the ATM networks depend on the implementation of the switches. A typical speed between the switch and a single workstation is 25 Mb/s, and between the switches 155-625 Mb/s. If the network is planned to be used for F-PRAM parallel computing, the upper-level switch should have enough bandwidth to deliver nearly all messages from the lower switches. We shall consider a $8 \times 7 = 56$-processor network consisting of clusters of 7 processors connected via a 25 Mb/s link to each of the 8 lower level switches, which are connected via a 155 Mb/s link to an upper level switch. On average, 7/8 of the messages of each of the 7 connected processors go to the upper level switch when emulating shared memory. From each of the lower levels to the upper level goes thus $7 \times 7/8 \times 25$ Mb/s = 153 Mb/s, which fits to the 155 Mb/s per connection bandwidth of the upper switch. Consequently, each of the processing nodes sees the 25 Mb/s ≈ 59 kpackets/s global bandwidth. For the 100 MOPS workstation, we get $B \approx 1700$, which is not much larger than the $B$ of the weakest parallel computers we presented in the previous sections. The latency of the ATM networks depend heavily on the implementation of the switches and the ATM interface cards of the workstations. Furthermore, the F-PRAM parameters depend on the shared memory emulation software used in each of the nodes. Consequently we shall not give here any values for the latency, or any other parameters.

**Block references**

Since delivering the standard 2-word long F-PRAM packets in workstation networks is not very efficient, the block references, which we defined in Subsection 4.3.2, are very useful for the parallel computations in the workstation networks. The block communication delay $L + k \times B_B$ models, e.g., the Ethernet network rather well. Using previous values with 64-bit words, we get $B_B = 12{,}800$, which is much better than the earlier $B = 2{,}000{,}000$. In practice, using packets having about a hundred words, we can exploit nearly the full bandwidth of the Ethernet. The hundred word requirement is not very restricting since most computationally intensive data parallel applications include rather long vectors. In an ATM network the advantage of the blocking of the references is not equally essential, but it helps. Using the 64-bit words, we get $B_B = B/5$ since we can deliver five words in each packet at cost of one word. For the ATM network of the above example, we get $B_B \approx 340$.

## 8.5 A sketch of an experimental FPM implementation

To make more accurate estimations on the values of the F-PRAM parameters in different parallel computers, we should measure them. To gain the full performance of each parallel computer, we would need to optimize each program separately for the architecture. This approach is not our goal. Instead, we are interested in testing how well the current parallel computers would execute our F-PRAM programs. This can be answered by evaluating the F-PRAM parameters for the computers using the F-PRAM primitives instead of the native primitives of the computers. Of course, the F-PRAM primitives have to be implemented using the native primitives, but in any case we want to know the costs of the F-PRAM primitives.

To be comparable with our previous experiments, the implementation of the F-PRAM model should use futures for shared memory access and *par-do* statements for parallelism presentation. The actual language is not as important. The value of the FPM language and compiler presented in the previous section was that we could emulate the execution easily using the F-PRAM emulator. Efficient and optimized compilation for real parallel machines would, however, require a much more complex compiler. Consequently, instead of trying to beat the existing state-of-the-art compilers, we should use them. More accurately, we could cross-compile the FPM source code to the native efficiently compileable language of the target machine. If the F-PRAM primitives can be efficiently expressed in terms of the efficient native primitives, the resulting implementation should be reasonably efficient.

### Matching of FPM to C with SHMEM libraries of the Cray T3E

As an example of the cross-compilation we sketch the compilation from FPM to the C with SHMEM (Shared memory routines) libraries of the Cray T3E. Compiling a procedural language to another is relatively easy. Especially compiling from Modula-2 to C seems to be very straightforward.

The SHMEM library of the Cray T3E includes *put* and *get* primitives to write and read data to and from the memory of another processing node. These can be used for implementing the *fwrite* and *future* statements of the FPM. Especially block references should result in efficient exploitation of the interconnection network [26]. The only missing feature is the asynchronous get. Due a design choice of the T3E processing nodes the processors wait idle the time while the network connection hardware takes care of the transferring the data from the remote memory to the local memory. In F-PRAM terms this would mean $B_P = L$ in case of futures. The *put* works asynchronously as the *fwrite* should. Remembering the difference in the impacts of latency and overhead in Section 7.1, the $B_P = L = 450$ is not very good. $L = 450$ does not impact the performance of the sorting a lot, but $B_P = 450$ affects an order of magnitude in execution time. The solution for this problem is to use larger block references to reduce the impact of long idling of the processor. The solution would not, however, be in par with the F-PRAM ideology. The lack of the asynchronous get was the reason why we abandoned our plans to do an experimental implementation in this thesis.

The SHMEM library does not provide a flat memory structure to exploit. Consequently, we have to implement the shared memory abstraction on top of the local memory systems. At the begin of the computation, each processor allocates a block of local memory to be used as a block of the shared memory. Each shared memory reference, more accurately address, is then transformed to a processing node number, local address pair using a suitable hash function. The choice of the hash function is probably the most important issue in an efficient implementation.

The parallelism handling method of our existing FPM implementation can be directly used in the C version. Each *par-do* statement of the FPM code can be implemented as a C version of Algorithm 6-2. As with the current implementation, we just have to maintain a stack of the values of $P$s and *PIDs*. Similarly, we have to introduce a predefined variable for each of the F-PRAM parameters. The variables are assigned at the initialization stage of the execution.

**Another alternative**

A more portable translation would be to use the direct remote memory access (DRMA) routines of the BSPlib [49]. The BSPlib includes routines *bsp_hpget* and *bsp_hpput* similar to the Cray SHMEM routines. Unfortunately, we feel that the goal of asynchronous, low overhead shared memory read, i.e., the future, cannot easily implemented with these routines either.

# Chapter 9

# Conclusions, critique, and future research

In the previous chapters we motivated and defined the new model of parallel computation called F-PRAM. In this chapter we draw conclusions about the results and especially sketch the future development of the F-PRAM model.

### Contributions

We have defined yet another parameterized computation model. The most important parameters do not differ much from the other existing models, such as the BSP and LogP. Compared to the existing parameterized models, however, we present some new features and a new combination of the existing features. The distinct features are

- data-oriented communication through the shared memory,
- use of the asynchronous futures for shared memory references,
- full asynchrony of the components (including the processors and the shared memory) of the computer, and a separate synchronization primitive, and
- a rather large set of secondary parameters for more accurate analysis.

We believe that these features provide a usable compromise especially for the programmer of parallel computers. Efficient concurrency of the computation and communication is very important for optimal performance. The shared memory based communication makes the data-oriented algorithms easier to write than using the process-oriented message passing models. The futures present a clean primitive for prefetching of the needed data. Compared to the other latency hiding methods, the futures require less processor employment and require no additional processor logic. Especially the asynchronous return of the future requests makes, e.g., tables of pending references needless within the processor. Also, as we do not require sequential consistency of shared memory references, the interconnection network may be easier to implement.

To support the computation model, we have defined a programming model and implemented it experimentally. This allowed us to write explicit executable programs for the F-PRAM model. The programming model supports the primitives of the computation models and aids the programmer to write portable programs.

To be able to study the impacts of the different parameters, we have designed and implemented a fully configurable emulator system of an abstract F-PRAM computer. By

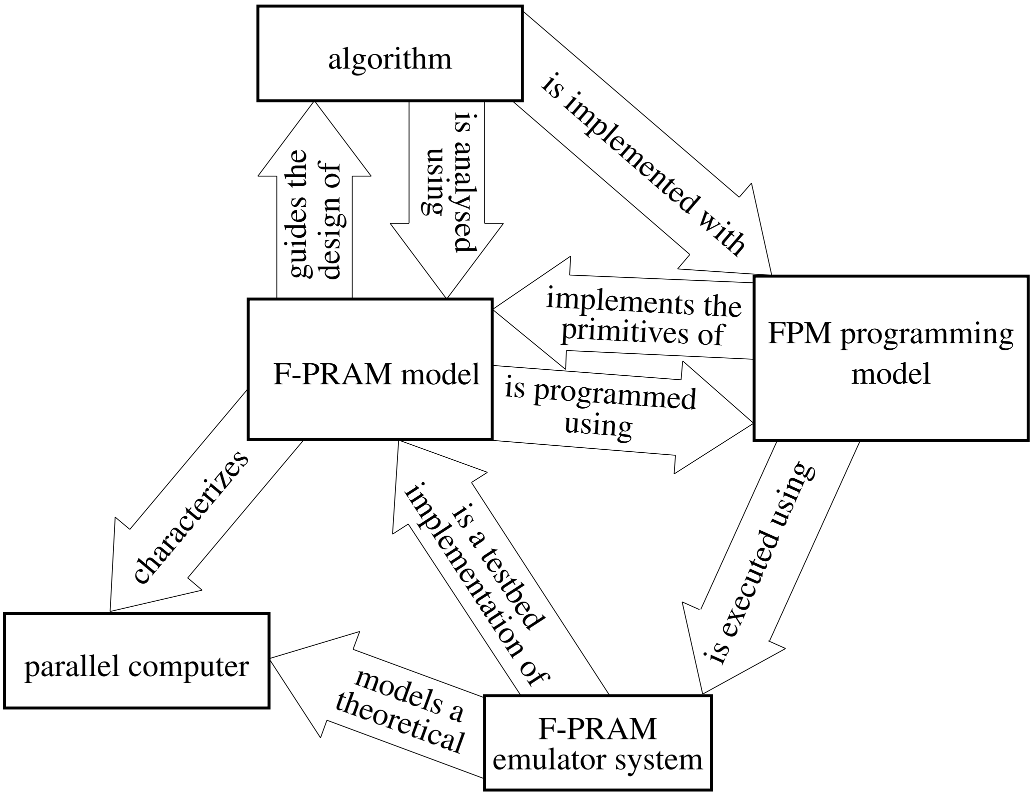


**Figure 9-1:** Relations of the key concepts of the F-PRAM model. Read in direction of the arrows.

using different configurations of the emulator system, we can study the impacts of the F-PRAM parameters without implementing the algorithm for different real parallel computer architectures. Especially we can study how much non-optimal features different parallel algorithms can tolerate without significant slowdown. The formal analysis on these constants of minimum requirements would be quite difficult.

Using the FPM language and our emulator system, we have implemented and analysed a set of example algorithms. Each of these examples has a distinct nature, and the results are thus different. In all cases we have been able to study the efficiency of parallelization in a form of speedup, and the impacts of the F-PRAM parameters. We have been able to present the limits of efficient execution in terms of the primary parameters $P$, $B$, and $L$.

To relate our model and the results of the measurements to the real world, we have studied some parallel computers in F-PRAM terms. Any accurate statements of parallel computers would require actual implementations, which would require access to the computers and significant programming efforts. Instead we made some estimations of the values of the F-PRAM parameters in some parallel computers in Chapter 8.

As a conclusion of this thesis, Figure 9-1 presents the relational schema of the key concepts and relations of the F-PRAM model. As the model itself, also the schema requires some streamlining to appear more appealing.

### Future research

As we have stated several times, all the current, and probably forthcoming, parallel computers have physically modularized shared memories. The current collection of measured results lack the data concerning the modularized shared memory and the simulated interconnection network. These are due to the shortage of these features in our experimental emulator system. We have not analytically studied the routing and congestion problems since they form another extensively studied complex problem. Consequently, our immediate future research should include rewriting of the emulation of the shared memory.

On the programming side we need more refined analysis of the use of the parameters $B_M$ and $M$ to avoid the congestions. Another important aspect of the shared memory performance is the fact that for most of the current computers the parameters $B_V$ and $B_M$ have values less than one, i.e., a single memory module or location cannot serve even one reference in one processor clock cycle. Consequently, to achieve good shared memory bandwidth, the parallel machine has to have more than $P$ memory modules. More formally, we need to analyse the optimal ratios of the values of the parameters $B$, $B_M$ and $M$. Moreover, the use of the parameter $B_B$, block transfer inefficiency, would have a considerable impact on the requirements of the number and the structure of the shared memory modules.

An interesting and useful research subject would be the usage of the F-PRAM cost model for the analysis of message passing algorithms. The cost model would need only minor changes, mostly on the secondary parameters. Furthermore, the message passing cost model would require a computation model which would define, for example, the message passing protocol. A practical implementation would consist of the inclusion of the F-PRAM parameters in the MPI message passing library.

# References

All URL addresses were valid at September 30th 1998.


[1] Agarwal A., Kubiatowicz J., Kranz D., Lim B.-H., Yeung D., D'Souza G., Parking M. 1993: Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, **13**,3, pages 48-61.

[2] Agerwala T., Martin J. L., Mirza J. H., Sadler D. C., Dias D. M., Snir M. 1995: SP/2 System Architecture. *IBM Systems Journal*, **34**,2, pages 152-184.

[3] Aggarwal A., Chandra A. K., Snir M. 1989: On Communication Latency in PRAM Computations. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 11-21. ACM Press, New York, NY.

[4] Ajtai M., Komlós J., Szmerédi E. 1983: An $O(n \log n)$ Sorting Network. *Combinatorica*, **3**,1, pages 1-19.

[5] Akl S. G. 1985: *Parallel Sorting Algorithms*. Academic Press, San Diego, CA

[6] Alexandrov A., Ionescu M. F., Schauser K. E., Scheiman C. 1995: LogGP: Incorporating Long Messages into LogP Model. *Proceedings of 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 95-105. ACM Press, New York, NY.

[7] Almasi G. S., Gottlieb A. 1994: *Highly Parallel Computing, 2nd ed.* Benjamin/Cummings, Redwood City, CA.

[8] Alverson R., Callahan D., Cummings D., Koblenz B., Porterfield A., Smith B. 1990: The Tera Computer System. In *Proceedings of 1990 International Conference on Supercomputing*, pages 1-6. IEEE CS Press, Los Alamitos, CA.

[9] Andrews G. R., Olsson R. A. 1993: *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, Redwood City, CA.

[10] ANSI 1983: *Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983.* Castle House, Turnbridge Wells.

[11] Arvind, Nikhil R. S. 1990: Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, **39**,3, pages 300-318.

[12] Bar-Noy A., Kipnis S. 1992: Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems. *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 13-22. ACM Press, New York, NY.

[13] Batcher K.E. 1968: Sorting Networks and their Applications. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307-314.

[14] Bilardi G., Herley T. K., Pietracapina A., Pucci G., Spirakis P. 1996: BSP vs. LogP. *Proceedings of 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 25-32. ACM Press, New York, NY.

[15] le Boudee J.-Y. 1992: The Asynchronous Transfer Mode: a tutorial. *Computer Networks and ISDN Systems*, **24**,4, pages 279-309.

[16] Blelloch G. E. 1989: Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, **38**,11, pages 1526-1538.

[17] Blelloch G. E., Gibbons P. B., Matias Y., Zagha M. 1997: Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors. *IEEE Transactions on Parallel and Distributed Computing*, **8**,9, pages 943-958.

[18] Bosch M., Franziskus S. 1994: *SB-PRAM simulator.* Universität des Saarlandes, Computer Science Department.
(http://www-wjp.cs.uni-sb.de/projects/sbpram/software.html).

[19] Brewer E. A., Dellarocas C. N., Colbrook A., Weihl W. E. 1991: *Proteus: A High-Performance Parallel-Architecture Simulator.* Technical Report MIT/LCS/TR-516. Massachusetts Institute of Technology.

[20] Callahan D., Smith B: *A Future-based Parallel Language for a General-purpose Highly-parallel computer.* Manuscript. (http://www.tera.com/).

[21] Chandra R., Gupta A., Hennessy J. L. 1994: COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, **27**,8, pages 14-26.

[22] Cole R. 1986: Parallel Merge Sort. *SIAM Journal of Computing*, **17**,4, pages 770-785.

[23] Cole R., Zajicek O. 1989: The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 169-178. ACM Press, New York, NY.

[24] Cormen T. H., Leiserson C. E., Rivest R. L. 1990: *Introduction to Algorithms*. MIT Press, Cambridge, MA.

[25] Cray Research 1995: *T3E Product Information*. (http://www.cray.com/).

[26] Cray Research 1997: *Cray T3E Optimization, TR-T3EOPT (D)*. Cray Research, Eagan, MN.

[27] Culler D. E., Dusseau A. C., Martin R. P., Schauser K. E. 1993: Fast Parallel Sorting Under LogP: From Theory to Practice. In *Portability and Performance for Parallel Processing*, pages 71-98. Wiley, New York, NY.

[28] Culler D., Karp R., Patterson D., Sahay A., Schauser K., Santos E., Subramonian R., von Eicken T. 1993: LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of 4th ACM Conference on Principles & Practices of Parallel Programming,* pages 1-12. ACM Press, New York, NY.

[29] Dongarra J. J. 1998: *Performance of Various Computers Using Standard Linear Equations Software.* Technical Report, University of Tennessee.

[30] Flynn M. J. 1972: Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, **21**,9, pages 948-960.

[31] Free Software Foundation 1998: *GCC - The GNU C Compiler (version 2.8.0).* (http://www.gnu.org/).

[32] Forsell, M. 1994: Are Multiport Memories Physically Feasible? *Computer Architecture News*, **22**,4, pages 47-54.

[33] Forsell, M. 1997: MTAC—A Multithreaded VLIW Architecture for PRAM Simulation. *Journal of Universal Computer Science*, **3**,9, pages 1037-1055.

[34] Fortune S., Wyllie J. 1978: Parallelism in Random Access Machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114-118. ACM Press, New York, NY.

[35] Fujitsu Ltd 1998: *VPP700E Product Information*. (http://www.fujitsu.co.jp/).

[36] Gajski D. D., Padua D. A., Kuck D. J. 1982: A Second Opinion on Data Flow Machines and Languages. *Computer*, **15**,2, pages 58-69.

[37] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V. 1993: *PVM 3.0 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Report TM-12187.

[38] Gibbons P. B. 1989: A More Practical PRAM Model. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 158-168. ACM Press, New York, NY.

[39] Gibbons P. B. 1996: What Good are Shared-Memory Models? *1996 International Conference on Parallel Processing, Workshop on Challenges in Parallel Processing*, pages 103-114.

[40] Gibbons P. B., Matias Y., Ramachandran V. 1994: Efficient Low-Contention Parallel Algorithms. In *Proceedings of 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 236-247. ACM Press, New York, NY.

[41] Gottlieb A., Grishman R., Kruskal C. P., McAuliffe K. P., Rudolph L., Snir, M. 1983: The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, **32**,2, pages 175-189.

[42] Halstead R. H. 1985: Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, **7**,4, pages 501-538.

[43] Harary F. 1972: *Graph theory.* Addison-Wesley, Reading, MA.

[44] Hayes J. P., Mudge T. N., Stout Q. F., Colley S., Palmer J. 1986: Architecture of a Hypercube Supercomputer. In *Proceedings of the International Conference in Parallel Processing*, pages 653-660.

[45] Hennessy J. L., Patterson D. A. 1990: *Computer Architecture: a quantitative approach*. Morgan Kaufmann, San Mateo, CA.

[46] Heywood T., Ranka S. 1992: A Practical Hierarchical Model of Parallel Computation. *Journal of Parallel and Distributed Computing*, **16**,3, pages 212-232.

[47] High Performance Fortran Forum 1993: High Performance Fortran Language Specification. Version 1.0. *Fortran Forum*, **12**,4, Special Issue.

[48] Hill J. M. D. 1997: *BSP Cost Parameters, Sorted by Megaflof/s Rate, for the Oxford BSP Toolset*. Oxford University Computing Laboratory. (http://www.bsp-worldwide.org/).

[49] Hill J. M. D., McColl W. F., Stefanescu D. C., Goudreau M. W., Lang K., Rao S. B., Suel T., Tsantilas T., Bisseling R. 1997: *BSPlib: The BSP Programming Library.* Technical report PRG-TR-29-9, Oxford University Computing Laboratory.

[50] Hillis D. W. 1985: *The Connection Machine*. MIT Press, Cambridge, MA.

[51] Hoare C. A. R. 1978: Communicating Sequential Processes. *Communications of the ACM*, **21**,8, pages 666-677.

[52] Hoare C. A. R. 1985: *Communicating Sequential Processes*. Prentice-Hall, New York, NY.

[53] Hämäläinen P. 1992: *An PRAM Emulator.* University of Joensuu, Department of Computer Science, Report B-1992-1.

[54] INMOS Limited 1988: *occam 2 Reference Manual*. Prentice-Hall, New York, NY.

[55] Intel Corporation 1991: *Intel Paragon Product Information*. (http://www.intel.com/).

[56] Iverson K. E. 1962: *A Programming Language*. Wiley, New York, NY.

[57] Jájá J. 1992: *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA.

[58] Jájá J., Ryu K. W. 1996: The Block Distributed Memory Model. *IEEE Transactions on Parallel and Distributed Systems*, **8**,7, pages 830-840.

[59] Juurlink B. H. H., Wijshoff H. A. G. 1996: A Quantitative Comparison of Parallel Computation Models. *Proceedings of 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 13-24. ACM Press, New York, NY.

[60] Juvaste S. 1992: *An Implementation of the Programming Language pm2 for PRAM*. University of Joensuu, Department of Computer Science, Report A-1992-1.

[61] Juvaste S. 1996: *Reasoning of a Parallel Computation Model*. Licentiate Thesis, University of Joensuu, Department of Computer Science.

[62] Karp R., Ramachandran V. 1988: A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*. Elsevier, Amsterdam.

[63] Knuth D. E. 1997: *The Art of Computer Programming, Volume 1: Fundamental Algoritms, 3rd ed.* Addison-Wesley, Reading, MA.

[64] Knuth D. E. 1997: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd ed.* Addison-Wesley, Reading, MA.

[65] Kruskal C. P., Rudolph L., Snir M. 1985: The Power of Parallel Prefix. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 180-185. IEEE CS Press, Washington.

[66] Kruskal C. P., Rudolph L., Snir M. 1990: A Complexity Theory of Efficient Parallel Algorithms. *Theoretical Computer Science*, **71**,1, pages 95-132.

[67] Laudon J., Lenoski D. 1997: The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium. on Computer Architecture*, pages 241-251. ACM Press, New York, NY.

[68] Lawson C., Hanson R., Kincaid D., Krogh F. 1979: Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, **5**,3, pages 308-371.

[69] Leighton T. F. 1992: *Introduction to Parallel Algorithms: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, CA.

[70] Leppänen V. 1996: *Studies on the Realization of PRAM*. Ph.D. Theses, TUCS Dissertations No 3, Turku Centre for Computer Science.

[71] Maggs B. M., Matheson L. R., Tarjan R. E. 1995: Models of parallel computation: a survey and synthesis. *Proceedings of the 28th Hawaii International Conference on System Sciences, vol. 2,* pages 61-70.

[72] Martel C., Raghunathan A. J. 1994: Asynchronous PRAMs with Memory Latency. *Journal of Parallel and Distributed Computing*, **23**,1, pages 10-26.

[73] Mascarenhas E., Knop F., Rego V. 1995: ParaSol: a Multithreaded System for Parallel Simulation Based on Mobile Threads. *Proceedings of the 1995 Winter Simulation Conference*, pages 690-697. ACM Press, New York, NY.

[74] Mattson T., Henry G. 1998: An Overview of the Intel TFLOPS Supercomputer. *Intel Technology Journal*, Q1/1998.

[75] Melhorn K, Vishkin U. 1985: Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, **21**, pages 339-374.

[76] Message Passing Interface Forum 1995: *MPI: A Message-Passing Interface Standard, version 1.1*. (http://www.mpi-forum.org/).

[77] Message Passing Interface Forum 1997: *MPI-2: Extensions to the Message-Passing Interface*. (http://www.mpi-forum.org/).

[78] Metcalf M., Reid J. 1992: *Fortran 90 Explained*. Oxford University Press, Oxford.

[79] Miyoshi H. et. al. 1994: Development and Achievement of NAL Numerical Wind Tunnel (NWT) for CFD computations. In *Proceedings of Supercomputing '94*, pages 685-692. IEEE CS Press, Los Alamitos, CA.

[80] Natvig L. 1990: Logarithmic Time Cost Optimal Parallel Sorting is not yet Fast in Practice! In *Proceedings of Supercomputing '90*, pages 486-494. IEEE CS Press, Los Alamitos, CA.

[81] Natvig L. 1991: *Evaluating Parallel Algorithms, Theoretical and Practical Aspects*. Ph.D. Thesis, Norwegian Institute of Technology. NTH-Trygg, Trondheim.

[82] Nishimura N. 1990: Asynchronous Shared Memory Parallel Computation. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 76-84. ACM Press, New York, NY.

[83] OpenMP Architecture Review Board 1997: *OpenMP Fortran Application Program Interface 1.0*. (http://www.openmp.org/).

[84] Papadimitriou C., Yannakis M. 1988: Towards an Architecture-Independent Analysis of Parallel Algorithms. *Proceedings of 20th ACM Symposium on Theory of Computing*, pages 510-513. ACM Press, New York, NY.

[85] Papadopoulos G. 1995: Taking sides on the SMP/MPP/Cluseter Debate. Keynote speech in the *1st International EURO-PAR Conference*, LNCS 966, page 3, Springer-Verlag, Berlin.

[86] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P. 1988: *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge.

[87] Ranade A. 1991: How to Emulate Shared Memory. *Journal of Computer System Sciences*, **42**,3, pages 307-327.

[88] Rao S. B. 1992: *Properties of an Interconnection Architecture based on Wavelength Division Multiplexing*. Technical Report TR-92-009-3-0054-2, NEC Research Institute, Princeton.

[89] Rosenblum M., Bugnion E., Devine S., Herrod S. A. 1997: Using SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, **7**,1, pages 78-103.

[90] Rowell J. 1995: Intel Captures $46M TeraFLOP Supercomputer Development Contract. *HPCwire*, **4**,36.

[91] Shepherdson J. C., Sturgis H. E. 1963: Computability of Recursive Functions. *Journal of the ACM*, **10**,2, pages 217-255.

[92] Siegel H. J. 1990: *Interconnection Networks for Large-scale Parallel Processing, Theory and Case Studies, 2nd Ed*. McGraw-Hill, New York, NY.

[93] Silberman J. et. al. 1998: A 1.0 GHz Single-Issue 64b PowerPC Integer Processor. *1998 IEEE International Solid State Circuit Conference*.

[94] Silicon Graphics Inc. 1995: *Power Challenge Technical Report*. (http://www.sgi.com/).

[95] Skillicorn D. B. 1991: Models for Practical Parallel Computation. *International Journal of Parallel Programming*, **20**,2, pages 133-158.

[96] Strassen V. 1969: Gaussian Elimination is not Optimal. *Numerische Matematic*, **14**,3, pages 354-356.

[97] Sun Microsystems: *Sun HPC 10000 Product Information*. (http://www.sun.com/).

[98] Thinking Machines Corporation 1991: *The Connection Machine CM-5 Technical Summary.*

[99] de la Torre P., Kruskal C. 1991: Towards a Single Model of Efficient Computation in Peal Parallel Machines. In *Proceedings of Parallel Languages, Europe*, LNCS 505, pages 6-24, Springer-Verlag, Berlin.

[100] Tucker L. W., Robertson G. G. 1988: Architecture and Applications of the Connection Machine. *Computer*, **21**,88, pages 26-38.

[101] Uthus I., Dybdahl H. 1997: *Simulation of the BSP Model on Different Computer Architectures*. Manuscript. Norwegian University of Science and Technology, Department of Computer Science.

[102] Utsumi T., Ikeda M., Takamura M. 1994: Architecture of the VPP500 parallel supercomputer. In *Proceedings of Supercomputing '94*, pages 478-487. IEEE CS Press, Los Alamitos, CA.

[103] Valiant L. G. 1990: General Purpose Parallel Architectures. In *Handbook of Theoretical Computer Science*, pages 943-971. Elsevier, Amsterdam.

[104] Valiant L. G. 1990: A Bridging Model for Parallel Computation. *Communications of the ACM*, **33**,8, pages 103-111.

[105] Veräjäntausta J. 1998: *An F-PRAM Emulator*. University of Joensuu, Department of Computer Science, Report B-1998-2. (ftp://cs.joensuu.fi/).

[106] Vishkin U. 1992: A Case for the PRAM As a Standard Programmer's Model. In *Proceedings of Parallel Architectures and Their Efficient Use*, pages 11-19.

[107] Vitányi P. M. B. 1988: Locality, Communication, and Interconnection Length in Multicomputers. *SIAM Journal of Computing*, **17**,4, pages 659-672.

[108] Vitter J. S., Simons R. A. 1986: New classes for parallel complexity: a study of unification and other complete problems for P. *IEEE Transactions on Computers*, **35**,5, pages 403-418.

[109] Wulf W. A., Levin R., Harbison S. P. 1981: *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, NY.

# Dissertations at the Department of Computer Science

**Rask, Raimo**. Automating Estimation of Software Size During the Requirements Specification Phase - Application of Albrecht's Function Point Analysis Within Structured Methods. Joensuun yliopiston luonnontieteellisiä julkaisuja, 28 – University of Joensuu. Publications in Sciences, 28. 128 p. Joensuu, 1992.

**Ahonen, Jarmo**. Modelling Physical Domains for Knowledge Based Systems. Joensuun yliopiston luonnontiteellisiä julkaisuja, 33 – University of Joensuu. Publications in Sciences, 33. 127 p. Joensuu, 1995.

**Kopponen, Marja**. CAI in CS. University of Joensuu, Computer Science, Dissertations 1. 97 p. Joensuu, 1997.

**Forsell, Martti**. Implementation of Instruction-Level and Thread-Level Parallelism in Computers, University of Joensuu, Computer Science, Dissertations 2. 121 p. Joensuu, 1997.

**Juvaste, Simo**. Modeling Parallel Shared Memory Computations, University of Joensuu, Computer Science, Dissertations 3. 190 p. Joensuu, 1998.